DECEMBER 1987 VOL. 10 NO. 4

a quarterly bulletin of the
Computer Society of the IEEE
technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON DATABASES AND LOGIC

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering . Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Databases and Logic

Foreword by the Guest Editor.

The reasons for the recent surge of interest in the area of Databases and Logic go beyond the theoretical foundations that were explored by early work in deductive databases, and include the following three motivations:

a) The projected future demand for Knowledge Management Systems. These will have to combine inference mechanisms from Logic with the efficient and secure management of large sets of information from Database Systems.

b) The realization that Logic supplies a natural means to extend the query languages of Relational systems into application development languages. This extension emancipates the development of database applications from the host language and solves the impedance mismatch problem currently besetting the development of database applications.

c) The realization that database applications supply a promising new application domain for Logic Programming.

Therefore, it is not surprising that, during the last three years, a number of projects were undertaken and numerous contributions were made by researchers coming mostly from the fields of Databases, Deductive Databases and Logic Programming. Because of space limitations, many of these contributions have not been included in this special issue that is intended as an introduction to the topic and a short description of some more significant endeavors.

The first paper in this issue, which describes the work being pursued at ECRC in Munich, also provides a short overview of the topical problems in this area. A first way of combining the power of Logic Programming with that of Databases consists in enhancing Prolog with database facilities; this is the approach described in the second paper. An alternate way consists in interfacing a Prolog front-end to a database back-end; the third paper illustrates the Prolog code optimization techniques used to support database queries efficiently in this environment. The remaining three papers follow a database-oriented approach which attempts to extend the execution mechanisms of relational database systems to handle the richer expressive power of Logic Programming languages.

Carlo Zaniolo

LOGIC APPROACH TO KNOWLEDGE AND DATA BASES at ECRC

Hervé Gallaire and Jean-Marie Nicolas

ECRC, Munich, FRG.

The European Computer-Industry Research Centre is a joint research organisation founded in January 1984 on the initiative of three major European manufacturers: Bull, ICL and Siemens. ECRC has been assigned the task to conduct precompetitive research in the field of Computer-Assisted Decision-Making and, presently, its research activity is centered around four main themes: Logic Programming, Knowledge and Data Bases, Human Computer Interaction and Computer Architectures for symbolic processing. The purpose of this paper is to give a brief overview of the studies presently underway in the field of Knowledge and Data Bases. The reader interested in a more technical presentation of the results obtained in this context is referred to the publications given in reference.

1. The Objectives and the Approach.

The objective of the Knowledge Base group is to contribute to the emergence and to the development of the technology allowing the construction of systems permitting an **intelligent** and **efficient** manipulation and control of large sets of information, regardless of whether this information is factual or general knowledge. The idea is to develop systems which combine the high-level information modeling features, the deductive capabilities and the flexibility of AI-based systems together with the efficiency and the control facilities provided by database systems over large sets of information [4]. It is worth emphasizing that such systems have to be viewed not only as extended Database Management Systems (DBMS), but also as Knowledge Programming Systems to be used as high-level programming tools in the development of knowledge-based systems.

Our approach to the development of Systems as introduced above, relies on a smooth integration of Artificial Intelligence and Data Base technologies, along the lines which have been drawn up by research on deductive databases [13]. This approach to KBMS building is now a common one (e.g. see [7]). This is not fortuitous. Indeed, it is in the AI world that one finds the inference mechanisms that provide the bases for an intelligent manipulation and control of information

and it is in the DB world that one finds efficient and secure management techniques for large sets of information.

2. Current Projects Overview

Current projects focus on four main topics which are investigated both on a theoretical and on a practical basis via system prototypes construction:

- **deductive knowledge manipulation**, i.e. how to combine inference techniques and database querying techniques so that deductive manipulation of large knowledge bases, stored on secondary devices, becomes computationally tractable (projects **EDUCE** and **DEDGIN**);

- **conceptual knowledge modeling**, i.e. how to logically represent, structure, and organize knowledge in a powerful and natural way (project **KB2**);

- **physical knowledge representation**, i.e how to physically represent, structure, organize and access knowledge efficiently (project **BANG**);

- **control of knowledge validity**, i.e. how to efficiently determine whether a set of rules is non-contradictory (project **SATCHMO**) and whether a knowledge base update violates a set of integrity constraints (project **SOUNDCHECK**).

Although dealing with different technical issues, these projects are in fact strongly interconnected as will be seen below. Further, they are organized in such a way that resulting prototypes can be used as tool kits in the development of various types of advanced information processing systems.

2.1. EDUCE

This project addresses, essentially from an engineering point of view, the problem of defining efficient cooperation schemes between inference systems and DBMSs. More precisely it focuses on various kinds of connections which can be set up between the inference system (and programming language) PROLOG and a relational DBMS. Two kinds of such connections have been identified as worthy of development: **a loose coupling**, which logically consists of an embedding of the database query language (e.g. the standard SQL or QUEL) into PROLOG and is physically implemented while providing an appropriate channel of communication between the two systems, run as two distinct processes; and **a tight integration** where the low level access mechanism of the DBMS is nested in the PROLOG system. Although these two types of connections are often considered as two alternatives for a PROLOG/DBMS cooperation scheme, the EDUCE system

prototype we have implemented supports both of them in a mixed way, so that **they can cooperate** [2, 3]. Further, EDUCE supports both (PROLOG) **clauses** and facts on disk and handles fully and efficiently PROLOG recursive clauses. Depending on the emphasis which is put on either of the two connections it supports, EDUCE can be viewed and used in two different ways. It is a DBMS extension in the sense that it provides the database application programmer with PROLOG as host language for the DBMS query language. It is a logic/database programming system in the sense that it permits the knowledge engineer, say, to write very large PROLOG programs whose clauses are efficiently managed on secondary devices. It is used in precisely that way for the development of the DEDGIN and KB2 prototypes (see below).

The initial version of EDUCE was developed based on the MU-PROLOG interpreter from the University of Melbourne and on the INGRES DBMS from the University of California/Berkeley. Other versions have also been developed (either by ECRC or its parent companies), thus permitting checking the applicability of the EDUCE connection schemes in different contexts (PROLOG compilers such as ECRC-PROLOG or IF-PROLOG and another DBMS, INFORMIX). Experiments with these various versions have shown that the smoothness of the logic programming/database cooperation as provided by EDUCE could be further improved, provided the PROLOG component be tuned for that and offers appropriate hooks for KB extensions [5]. This aspect of the problem is currently being tackled in the context of the SEPIA (Standard ECRC Prolog Integrating Advanced features) compiler [6] under development. The project will then evolve in two directions: first, realization of two upgraded EDUCE versions involving SEPIA as a PROLOG component and two full-fledged commercial DBMSs (ORACLE, INGRES); and second, connection of SEPIA with the dedicated file sytem BANG, as a step towards a persistent logic programming system.

2.2. DEDGIN

This second project is also concerned with the exploitation of a smooth cooperation between inference and database querying techniques, but from a different perspective and with different objectives. Whereas EDUCE exploits such a cooperation in the context of a **logic programming language** to achieve a (logic/database) programming system, the purpose of the DEDGIN project is to **enhance the query/answering component of a DBMS** with an appropriate deductive unit, so as to obtain a **Deductive Database System** where both base and derived relations can be efficiently queried by a **casual database user**. On the one hand, this means that the user has to be provided with a purely declarative language (e.g. datalogic) for expressing derived relation definitions (i.e. derivation rules) and with, say, an SQL-like language for querying; but on the other hand it also means that any problems related to deductive query evaluation must be hidden from the user and tackled fully by the system. A major technical issue here is recursion handling, which occurs when derived relations are defined, either directly or indirectly, in terms of themselves.

The present version of DEDGIN [18] has been implemented (via EDUCE) as an extension of the INGRES relational DBMS. Its deductive query evaluator is based on an algorithm, called QSQ for Query-SubQuery [17], which properly and efficiently handles recursive definitions. For any kind of recursion, it guarantees the termination of the query evaluation process, it ensures answer completeness and restricts access to those database facts which are effectively needed to answer the query. If various methods for recursive query evaluation have been proposed in the literature (e.g. see [1] for an overview), only a few of them are general in the sense that they possess the above-mentioned properties for any kind of recursion and any data configuration. Although QSQ is general, it is efficient. But better performances may be obtained for some specific classes of recursion. More recent work for obtaining a logic-based theoretical framework for recursive query processing has led to the definition of the database-complete proof procedure SLD-AL [19]. This proof procedure also provides an appropriate support for comparing the various methods available for recursive query evaluation and for introducing the notion of "Global Optimization" which properly incorporated into QSQ, has led to another method named QoSaQ [20]. The global optimization technique enables QoSaQ to remain a general method and yet to perform on specific classes of recursion as least as well as specialized methods specifically designed for these classes. The QoSaQ method is being investigated in detail and will be taken into account in the design of the next version of the system.

2.3. KB2

Efficient deductive manipulation of large sets of knowledge is one but not the only key issue in building KBMSs. Powerful information modeling features and efficient semantic integrity enforcement mechanisms are also needed. The present version of the system embodying these concepts and features is called KB2 [21]. It is developed in EDUCE. KB2 offers a semantic information model which is roughly an Entity-Relationship model **extended** by adding the "generalization" logical structuring feature (IS-A hierarchy in the AI parlance) and with a logic-based assertional language for expressing integrity constraints as well as general axioms defining derived relationships. Its integrity subsystem, named SOUNDCHECK [10], implements a constraint enforcement technique we have defined, which reduces as much as possible the set of information to be checked for controlling update validity. Besides the above-mentioned characteristics, an essential property of the KB2 system lies in its versatility and flexibility of use. This is the result of adequate schema manipulation facilities and a smooth interconnection between the implementation language (PROLOG/EDUCE), the (logic-based) query and rule definition languages, and the knowledge manipulation language (PROLOG again). Various ECRC applications are currently being developed while using the KB2 prototype, the feedback from which should lead to an extended version of the system.

2.4. BANG

Most of the physical data structures used at present in DBMSs are such that access to a file via more than one key involves multiple indices with associated update overheads. Such structures may be adequate to support menu-driven user interfaces, for which the form of the allowed queries is entirely controllable and predictable, but they are a very poor match for an interactive query language. Such a language gives the user much more freedom and flexibility, but there is no longer any guarantee that a query will match the underlying organization of the data structures. In a conventional database the experienced user can at least learn what kind of questions not to ask, i.e. the questions that always seem to take a very long time to answer. However, in a knowledge base or a deductive database, the conceptual relationship between the query and the underlying data structures may be completely hidden by a host of intermediate deduction rules. The user still initiates the query, but has no control over the course which it will take. Hence the need to find more appropriate data structures which must be designed so that the response time to a query should depend essentially on the complexity of the query and not so much on the particular combination of attributes named in it.

Towards that end, and with the immediate objective to substitute more powerful and flexible data structures for the INGRES structures on which the system EDUCE and thus - indirectly - KB2 and DEDGIN presently rely, the potential offered by multi-dimensional file organization is being investigated. A new file structure, named the BANG file, has been devised [11, 12]. A "grid-file" type, it has the fundamental advantage over previous designs that its directory always expands at the same rate as the data, whatever the data distribution. Its implementation is underway and preliminary tests have given satisfactory results. The project will proceed with studies of the impact of the BANG file on techniques for query evaluation.

2.5. SATCHMO.

This project aims at the development of the kernel of a Computer-Aided Schema Design system for knowledge and/or databases, namely, a system that permits a knowledge or database administrator to represent, manipulate and control, in an interactive way, the structural information and rules which describe a particular KB or DB application. Besides standard information representation and manipulation facilities, such a system will offer as control facilities the possibility to check for the well-formedness of the information it manages. In particular it will check for the consistency (non-contradiction) of the set of rules. One of the key tasks in this project is therefore to design an appropriate formal method for rule consistency checking, to complement it with adequate strategies and to implement it in a procedure for automatic consistency checking [9]. One will note that such procedures are required not only in the context of systems assisting the schema design process, but also as part of the knowledge assimilation component of a KBMS as well as in any advanced information system.

At present, the basic functionalities of the design system have been developed and

significant results have been obtained for rule consistency checking. An original procedure, based on model generation, has been devised and implemented in PROLOG [14]. This procedure, named SATCHMO, is currently being tested and has shown satisfactory efficiency when run on some well-known benchmark examples recently discussed in the field of automated deduction. The project is further proceeding in two directions: first, development of more sophisticated strategies to be incorporated in SATCHMO, in particular, making use of techniques for integrity enforcement, like those designed for the SOUNDCHECK component of the KB2 system, in order to enhance the efficiency of the model generation process [8]; second, evaluation of the performance of the procedure when confronted with "real-life" problems.

3. Conclusion.

Results obtained so far in the context of the various projects briefly introduced above or in the context of associated studies [15, 16, 22]certainly do not permit to claim that the development of "Logic/DB based KBMSs" is presently fully mastered, but we consider them as significant enough for the development of a sound prototype of such a KBMS be undertaken and for claiming that:

- A programming tool coupling a deductive system (Prolog) and a database, such as EDUCE, is feasible and interesting in its own right. It has been shown to be effective, particularly when using the tight integration mechanisms, rather than the loose coupling ones: this is necessary as soon as complex queries, leading to embedded calls, in particular recursive ones, are made to the DBMS from Prolog. This is not to be taken as a surprise of course. But we emphasize that even the loose coupling connection can be satisfactorily used in applications, and that the often made claim that tuple at a time (of Prolog) cannot work in a database context is not true in all contexts. Furthermore we have found that programming tools of the EDUCE kind definitely constitute a useful and performant basis for the implementation of knowledge-based systems like a Deductive Database Query-Answering system (e.g. DEDGIN) or a KB system (e.g. KB2). EDUCE is admittedly just a first step towards such a tool, based on the connection (coupling or integration) of two independent systems; the next step consists in the realization of a fully integrated logic/DB programming system.

- Deductively augmented relational DBMSs (e.g. DEDGIN) offering performances compatible with that of their underlying DBMS, can be developed. The underlying DBMS can be used in this context more efficiently than in the first type of system, since there is a treatment of the query (and of the programs used to answer it) in the system which allows to take advantage of the optimization techniques of the DBMS. Although progress can still be made in the future, performances obtained so far with DEDGIN provide enough evidences to support such a claim.

- Merging logic programming and semantic modelling features (i.e.

classification, generalization, aggregation...) as achieved in the context of the KB2 prototype, brings additional power to each individual paradigm. By merging the two techniques, one gets more than the mere sum of each technique. The ultimate goal pursued in trying to blend such techniques, is to bridge the gap between the semantic power of the programming approaches, both logic and object-oriented ones, the database approach (in particular the semantic models), and the AI approach (frames, semantic nets, ets). Not all conflicts have been resolved as our early experiments indicate, but progress is being made in this direction.

References

[1] Bancilhon, F., Ramakrishnan, R., "An Amateur's Introduction to Recursive Processing Strategies," *Proceedings ACM-SIGMOD'86 Conference*, May, 1986, Washington, D.C., U.S.A.

[2] Bocca, J., "EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS," *Proceedings 3rd IEEE Symposium on Logic Programming SLP'86*, Sept. 1986, Salt Lake City, U.S.A.

[3] Bocca, J.,"On the Evaluation Strategy of EDUCE," *Proceedings ACM-SIGMOD'86 Conference*, May 28-30, 1986, Washington, D.C., U. S. A.

[4] Bocca, J., Decker, H., Nicolas, J.-M., Vieille, L. and Wallace, M., "Some Steps Towards a DBMS Based KBMS," *Proceedings 10th World Computer Congress IFIP'86*, Sept., 1986, Dublin, Ireland.

[5] Bocca, J. and Bailey, P., "Logic Languages and Relational DBMSs: the Point of Convergence," *Proceedings Workshop on Persistent Object Stores*, Aug., 1987, Appin, Scotland.

[6] Bocca, J., Meier, M. and de Villeneuve, D., "The Specification of a Compiler with High Performances and Functionalities-SEPIA Prolog," ECRC TR-PC-1, May, 1987.

[7] Brodie, M., Mylopoulos, J., eds., *"On Knowledge Base Management Systems, Integrating Artificial Intelligence and Database Technologies,"* Springer-Verlag, 1986, New York, U.S.A.

[8] Bry, F., Decker, H. and Manthey, R., "A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases," ECRC TR-KB-16, Oct., 1987.

[9] Bry, F. and Manthey, R., "Checking Consistency of Database Constraints: a Logical Basis," *Proceedings 12th International Conference on Very Large Data Bases VLDB'86*, Aug. 25-28, 1986, Kyoto, Japan.

[10] Decker, H., "Integrity Enforcement on Deductive Databases," *Proceedings 1st International Conference on Expert Database Systems EDS'86,* April 1-4, 1986, Charleston, South Carolina, U.S.A.

[11] Freeston, M., "Data Structures for Knowledge Bases: Multi-Dimensional File Organisations," ECRC TR-KB-13, August, 1986.

[12] Freeston, M., "The BANG File: a New Kind of Grid File," *Proceedings ACM-SIGMOD'87 Conference*, May, 1987, San Francisco, U.S.A.

[13] Gallaire, H., Minker, J. and Nicolas, J.-M., "Logic and Databases: a Deductive Approach," ACM Computing Surveys, June, 1984,.

[14] Manthey, R. and Bry, F., "A Hyperresolution-Based Proof Proceedure and its Implementation in Prolog," *Proceedings German Workshop on Artificial Intelligence GWAI'87*, Sept. 1987, Geseke, Germany.

[15] de Rougemont, M., "Theory and practice of intensional compilation of queries," ECRC IR-KB-22, 1986.

[16] de Rougemont, M., "Constructive second-order proofs in logical databases," *Proc. 10th Int. Joint Conf. on Artificial Intelligence IJCAI'87*, Aug. 23-28, 1987, Milan, Italy.

[17] Vieille, L., "Recursive Axioms in Deductive Databases: The Query-Subquery Approach," *Proceedings 1st International Conference on Expert Database Systems EDS'86*, April 1-4, 1986, Charleston, South Carolina, U.S.A.

[18] Vielle, L., "Recursion in Deductive Databases: DEDGIN, a Recursive Query Evaluator," *AFCET Conference*, Sept., 1987, Sophia-Antipolis, France.

[19] Vielle, L., "Database-Complete Proof Procedures Based on SLD-Resolution," *Proceedings 4th International Conference on Logic Programming ICLP'87*, May, 1987, Melbourne, Australia.

[20] Vieille, L., "From QSQ towards QoSaQ: Global Optimization of Recursive Queries," ECRC IR-KB-42, January, 1987.

[21] Wallace, M., "KB2: a Knowledge Base System Embeded in Prolog," ECRC TR-KB-12, August, 1986.

[22] Wallace, M., "Negation by Constraints: a Sound and Efficient Immplementation of Negation in Deductive Databases," P*roceedings 4th IEEE Symposium on Logic Programming SLP'87*, Sept., 1987, San Francisco, U.S.A.

The NU-Prolog Deductive Database System

Kotagiri Ramamohanarao, John Shepherd, Isaac Balbin, Graeme Port,
Lee Naish, James Thom, Justin Zobel, Philip Dart

Department of Computer Science, University of Melbourne,
Parkville, Victoria 3052, Australia

Abstract

We describe the current state of the design and implementation of the NU-Prolog Deductive Database system. The ultimate aim of the NU-Prolog project is to produce a unified logic/database system, where programs may be expressed declaratively, in a form close to first-order logic, and at the same time efficiently access very large knowledge bases. The system addresses the problem of integrating a logic engine with a database system at a number of levels: query languages, indexing schemes to efficiently access large quantities of data, efficient join facilities, novel computational methods for answering queries, and transaction processing for concurrent updates.

1. Introduction

At an abstract level, mathematical logic provides a uniform framework for the expression and manipulation of information. One of its greatest strengths, from the point of view of computer science, is that the manipulation of information can be given a semantics which is declarative. That is, the semantics can be expressed without reference to a sequence of operations. Research in the field of Logic Programming is concerned with developing logic-based programming systems which manipulate data efficiently. Prolog is a logic programming language which has been successfully used as a general programming language.

Techniques have been developed for traditional database query systems such as SQL to manipulate large amounts of information very efficiently. The way information is handled in these systems can be expressed within a subset of logic. These systems typically allow the retrieved information to be transformed using a fixed set of operations, but fall short of providing a general computational mechanism for transforming data; that is, they do not provide the fundamental logical operation of *deduction*. The introduction of deduction into traditional databases has lead to creation of the new field of *deductive databases* [9]. At a semantic level these are equivalent to logic programs but at an operational level they are quite different. A query to a logic program is generally computed top-down while a query to a deductive database is generally computed bottom-up. These two computation methods are the extremes of the range of computation methods that might be employed by a deductive database system.

This paper reports the developments so far in integrating a database facility with the NU-Prolog system [24]. The investigation has proceeded at several levels: from low level database indexing schemes; through mixed computation methods for programs containing recursion and negation; to high level query optimisation by program transformation. At the user-interface level, NU-Prolog provides a query language based on the syntax of first order logic. The ultimate aim of the NU-Prolog project is to develop a unified logic/database system which retains the logic programming style, and can efficiently access predicates consisting of very large numbers of facts. The system presents a uniform interface and provides efficient access to all predicates in the system, whether they are stored in main memory as part of the logic engine, or whether they reside on secondary storage in the database system.

Our philosophy is to exploit existing techniques wherever possible. At the lower levels of the system, techniques such as tuple indexing and join processing developed for relational databases have been adapted for use in the NU-Prolog system. At higher levels, there are a number of possible computational methods and it is still an active area of research to determine which method is the most effective in a given situation. In all of these methods we can make use of standard query optimisation techniques such as unfolding, common subexpression elimination and reordering of goals.

In this paper we review the methods used by the NU-Prolog deductive database system to solve several important problems. An overview of the user interface of the system (the NU-Prolog query language) is given in section 2. Section 3 describes our clause indexing scheme for providing efficient access to large databases. The connection between the database system and the logic engine is examined in section 4. Section 5 describes an efficient join operation which exploits our clause indexing scheme. Section 6 examines bottom-up computational methods which we have developed for recursive query processing on deductive databases. A technique for handling concurrent updates to deductive databases based on transaction processing is detailed in section 7.

We are also working on a number of outstanding problems in deductive database systems. We are developing methods for evaluating queries to recursive database programs containing negation using a mixture of top-down and bottom-up computation. We are designing a uniform intermediate language to compile the database into, which is more amenable to direct execution by the lower levels of the system. We are investigating issues pertaining to the choice and definition of sideways information passing strategies.

1.1. Terminology

Deductive databases have their origins in two traditionally distinct fields of computer science: logic programming and databases. These fields have developed their own terminologies, but deal with similar concepts. This has lead to a situation where there is a confusing mixture of notations in use and so we devote the remainder of this section to defining the notation used in this paper. (The reader who is interested in further tracing the development and terminology of the deductive database field will find a comprehensive set of references in [1].)

The basic components of logic programs, as defined in [12], are: *variables*, *constants*, *function symbols*, and *predicate symbols*. We adopt the Prolog convention of denoting variables by strings of characters beginning with an upper case letter; constants, function symbols and predicate symbols (*relation names*) are denoted by strings of characters starting with a lower case letter.

A *term* is a variable, a constant, or a function symbol applied to other terms as in f(t1,t2,...,tn). An *atom* is of the form p(t1,t2,...,tn), where p is a predicate symbol and t1,t2,...,tn are terms. A *literal* is either an atom, or an atom preceded by $\neg$, the negation sign. A *rule* (*clause*) is a statement of one of the forms

p :- q1,q2,...,qn. or p.

p is an atom and q1,...,qn are literals. p is called the *head*, the conjunction q1,q2,...,qn is called the *body*, and each qi is a *body literal*. A rule of the form p. is also known as a *unit clause*. Without loss of generality, a *query* a, where a is an atom, is a statement of the form ?-a. A *goal* is a conjunction of literals, where each literal is called a *sub-goal*.

An atom or term is ground when it contains no variables. <t1,t2,...,tn> is known as a *tuple* of p. A *fact* is a ground unit clause. A *base predicate* is a predicate defined solely by facts. A *derived predicate* is a predicate which is not defined solely by facts. A *database* is an (unordered) set of rules. We use the terms database and program interchangeably.

2. The NU-Prolog Query Language

A major aim in the design of the NU-Prolog language is to allow programs to be written more logically than is possible with other versions of Prolog; NU-Prolog is a step towards purely declarative logic programming. The body of a NU-Prolog rule and goal can take the form of an arbitrary first order formula. In addition, NU-Prolog provides aggregate functions, and a declarative if-then-else construct. The NU-Prolog user interface makes the NU-Prolog language available as a query language with facilities for the multikey sorting of answers, and the concise specification of insertions, deletions and updates.

Lloyd and Topor have shown how a program specified using arbitrary first order formulae can be transformed to a program with clause bodies consisting of a conjunction of literals [11]. NU-Prolog provides both existential and universal quantification using the binary prefix operators some and all in the form Quantifier Variables Formula, to quantify the variables in the term Variables over the scope Formula. Conjunction, disjunction, implication are provided by the five infix operators ,, ;, =>, <=> and <= used in the form Formula Operator Formula. Negation is provided in the form not Formula. Five aggregate functions are currently provided: solutions, count, max, min and sum; others can be easily added.

NU-Prolog supports queries of the form Head : Body where Head is a term and Body is a NU-Prolog formula. Different forms for the head of a query are recognised. If the head of a query is empty, the query is answered by printing either Yes or No. If the head of a query is of the form Term sorted or Term sorted Keylist, then answers to the query will be sorted or multikey sorted, with duplicates removed. Sort keys can be specified with one of the prefix operators + or – to indicate ascending or descending order. Answers to queries are printed in the form of Term (or Head if sorted is not specified) with the bindings generated by running the body of the query. For example, a query could be run as follows:

```
?- A/B  sorted +A, -B  :  member(A/B, [2/3, 4/5, 2/6]).

2/6
2/3
4/5
```

Insertions can be performed with a statement of the form insert Termlist where Body. For each of the bindings generated by running the body of the insertion, each of the terms in Termlist will be asserted. For example, the following insertion will create a unary relation of the suppliers who supply only red parts.

```
?- insert redSupplier(S) where supplier(S), all P supplies(S,P) => part(P,red).
```

Deletions can be performed with a statement of the form delete Termlist where Body. For each of the bindings generated by running the body of the deletion, in conjunction with the terms of Termlist, each of the terms in Termlist will be removed from the database. For example, the following deletion will delete the suppliers who don't supply any red parts.

```
?- delete supplier(S) where not (some P (supplies(S,P), part(P,red))).
```

Updates can be performed with a statement of the form update Updatelist in Atomlist where Body, where Updatelist consists of terms of the form Var to Atom. For each of the bindings generated by running the body of the update, in conjunction with the terms of Atomlist, each of the terms in Atomlist will be retracted, and asserted with Var replaced by Atom. For example, the following update will change the colour of any yellow part to red.

```
?- update C to yellow in part(P,C) where C = red.
```

The where Body component of an insertion, deletion or update may be omitted; this is equivalent to using where true.

These facilities, in combination with the history mechanism and command editing facilities of the NU-Prolog user-interface, combine to provide a powerful, yet concise database query language.

3. Indexing Schemes for Efficient Record Access

For databases used in real-world applications, a base predicate is often defined by a large number of facts (possibly in the millions), while the number of rules defining each derived predicate is generally quite small. Since derived predicates are small and accessed frequently during a computation, they are most efficiently stored within the Prolog system; that is, as part of the *internal database*. Since the base predicates are possibly very large, they reside on secondary storage as part of the *external database*, and are brought into the Prolog system only when required. However, no matter how a predicate is physically stored, we need indexing schemes to efficiently locate the subset of facts/rules in a predicate which are "relevant" to answering a query on that predicate.

The above characterisation of base predicates as external database predicates and derived predicates as internal database predicates is not clear cut. There may be applications, such as expert systems, where derived predicates are very large, and it is more efficient to store them as part of the external database. Some of the indexing schemes described in this section therefore allow variables as arguments to external database predicates.

A query which has values specified for a subset of its arguments is known as a *partial-match query*. The execution of a partial-match query on a predicate selects the subset of tuples in the predicate that have the specified values for the ground arguments. An effective technique for answering partial-match queries employs superimposed codewords The NU-Prolog deductive database system uses a variant of these schemes [21] for the storage and retrieval of Prolog terms. This is a generalisation of previous schemes (such as [22]) as it allows indexing on terms containing variables and function symbols in the external database.

In a superimposed codeword (*simc*) indexing scheme each fact in the database has an associated *descriptor* – a bit-string formed by superimposing (bitwise OR-ing) the *codewords* for the individual arguments of the fact. A codeword is the hash value of a ground argument. One possible method of codeword generation is to use the ground argument as a seed for a random number generator which indicates the bit positions to be set to one. In order to determine which facts satisfy a partial-match query, the descriptor for the query (formed by OR-ing the codewords for the non-variable arguments) is AND-ed with the descriptor for each fact in the database. The bits set to one in the query descriptor are a subset of the bits set to one in the descriptor of any matching fact, and so the AND operation will result in a bit-string which is identical to the query descriptor for each matching fact. While examining the descriptors is clearly more efficient than scanning the records themselves, it can be further improved by using a two-level descriptor scheme. In such a scheme, the database is divided into segments (pages), corresponding to pages on the physical device. Each fact is allocated to a segment at insertion time and a descriptor is maintained for each segment, derived from the descriptors of the facts in the segment. We scan the segment descriptors to determine which segments might contain matching facts, and then scan the fact descriptors only in those segments.

Standard *simc* indexing schemes assume that all the unit clauses in the external database are facts, that is, contain no variables. In addition, they treat arguments, such as g(1), as strings of characters without considering their internal structure. Since we want to allow arbitrary unit clauses in the external database, our indexing scheme must be able to support variables and function terms.

In order to represent variables in unit clauses (ignoring terms with function symbols for the moment), we append *mask bits* to the end of each descriptor to denote which arguments are ground. Each bit in this *mask* is associated with a particular argument in the unit clause. A mask bit is set to zero when the corresponding argument is ground. When the mask bit is set to one, the corresponding

argument is a variable (and consequently contributes no bits to the descriptor), and any non-matches on this argument are ignored. The matching process now becomes somewhat more complicated: for each argument, if the associated mask bit in either the query descriptor or the fact descriptor is set to one, we treat that argument as a match, regardless of codewords. If both mask bits are zero, we compare the codewords of the argument in the query and the fact to determine whether there is a match on that argument. If all arguments successfully match, then we have found a potentially matching fact.

In representing the structure of a function term, we consider the term first as a single argument to determine the total number of bits which it provides in the descriptor. We then partition the bits among the components of the term according to a user-defined weighting scheme, and generate the codeword for the entire term by superimposing the codewords for the components. To handle variables within a term, we require the database designer to specify the maximum *indexing complexity* of the term, that is, the maximum number of components in the term which will be used for indexing purposes. Given such a specification, we allocate one mask bit for each component. If a term is less complex than the specification indicates, we simply ignore the mask bits which were allocated for the "unused" components. If the term is more complex than the specification, we prune it; that is, we ignore any components "outside" the specification. Note that the entire term is always stored in the external database; parts of the term are "ignored" only for indexing purposes.

To store rules in the external database, a rule such as p(X,Y):-q(X,Z),r(Z,Y) may be represented as a unit clause :-(p(X,Y),(q(X,Z),r(Z,Y))). The indexing scheme would use only the first argument, since we only match with the head of the rule. However, this representation stores all rules in one predicate (the :- predicate) and NU-Prolog programs typically contain large numbers of rules with considerable variation in the arity and complexity of arguments in the head. It is thus necessary either to define a very general structure for the head, most of which will usually be wasted, or to lose indexing information due to pruning when searching on rules with large heads. For better performance, all rules defining a derived predicate could be stored in a format such as p(X,Y,(q(X,Z),r(Z,Y))). After retrieval from the database, this must be transformed into the standard rule-form.

The *simc* indexing scheme efficiently determines which facts have a high probability of matching the given query. Consider, however, the example of a query which retrieves one hundred facts from a database containing a million facts. Each potential matching fact might come from a different segment (page) and the query may cost one hundred disk accesses. It is desirable to allocate facts to segments in such a way that facts which are likely to be retrieved together are stored in the same segment. We achieve this by employing a clustering scheme based on the distribution of query types [17-20].

In some applications, the number of facts in a relation may grow or shrink by orders of magnitude over the lifetime of the relation. We call such relations *dynamic*, to contrast them with *static* relations where the number of facts changes very little. The NU-Prolog deductive database system has two *simc* indexing schemes available, one to deal with (relatively) static relations, and one for dynamic relations of ground facts. The *simc* indexing scheme for static relations uses a bit-sliced organisation for storing descriptors to minimise the scanning of the descriptor file. Consequently, the user must specify an upper bound on the number of facts which are stored in a relation.[1] Several techniques have been proposed for indexing dynamic databases [10, 19]. All are based on clustering and, in particular, on multi-key hashing where each argument contributes a portion of the index (segment address) bits. The index bits are interlaced to form the final index but, unlike clustering in the static case, the number of bits allocated to each argument can grow and shrink as the number of facts in the database changes. In our dynamic *simc* indexing scheme (*dsimc*), the superimposed codeword descriptor serves the dual roles of a hash value and a segment address. Because of this, it is not possible to use mask bits to

[1]In fact, the database *can* be expanded, but only at the cost of massive reorganisation of the index and data files.

describe variables and all unit clauses in dynamic *simc* databases must be completely ground.

For good retrieval performance, all of the above methods depend on the data being approximately uniformly distributed. No effective algorithms are known for handling databases where the distribution of the data is non-uniform or the data is correlated.

4. Logic Engine/Database Interface

The top-down method is well suited to answer queries to predicates of the internal database. Since this is the computation method used by Prolog systems, we assume that the internal database is part of the logic engine. Answering queries to the external database using this method can be simply integrated with a Prolog system since the backtracking information placed on the call stack may refer to files on secondary storage as easily as to rules represented internally. The same operations are performed in both cases: create a stack frame when a query is first issued, advance a marker when backtracking and delete the stack frame when all answers have been obtained.

An important aspect of the interface is the mechanism by which tuples are transferred from secondary storage into the working memory of the NU-Prolog engine. In our implementation facts are retrieved from the database in textual form and "read" into working memory by the NU-Prolog reader. In addition, queries are sent to the database system using the NU-Prolog printing predicates. In the initial implementation of the NU-Prolog deductive database system, it was discovered that, in answering a query which returned a large number of answers, up to 80% of the computation time spent was spent inside the NU-Prolog reader, parsing the facts retrieved by the database system. The interface now employs its own parser to reduce this bottleneck by an order of magnitude.

In the future, we plan to investigate implementations where the database system uses the same internal representation of terms as the NU-Prolog engine. One way of achieving this is to map the database directly into the working memory of the Prolog system. Uniform virtual memory [8] ensures that the database part of virtual memory is shareable so that multiple processes can access and update the database concurrently. However, for large databases there are problems with loading and resolving references to the very large pieces of code required for the base predicates. This can be handled by using dynamic linking methods involving indirect/multiple symbol tables.

An advantage of textually-based interfaces is that they permit relatively simple integration of the logic engine with *existing* database systems. All that is required are routines to map between the representation of records as tuples and their representation as Prolog terms. We have followed this approach in connecting NU-Prolog to Unify's[2] SQL interface [27] so that the system can access existing Unify databases.

While the top-down method of answering queries to the external database is simple to implement and provides an effective solution for many queries, it is not the most efficient method in all cases. For conjunctions of queries, the top-down, tuple-at-a-time computation rule is equivalent to the nested loop method of computing joins which is not practical for real applications (that is, where the entire predicate cannot be stored in main memory). More importantly, the top-down computation rule has the disadvantage that it is not guaranteed to terminate; bottom-up methods, on the other hand, always terminate in the absence of function symbols. In addition, database queries are commonly of the 'return *all* answers' type, while the top-down method is more suited to the 'return *an* answer' type query. Thus it cannot automatically make use of well-established optimisation techniques. We discuss methods to overcome these problems in later sections.

[2] Unify is a trademark of the UNIFY Corporation.

5. The Superjoin Algorithm

A conjunction of Prolog goals which depends on only base predicates can be naturally viewed as a join operation. Although partial-match indexing significantly improves the efficiency of the join operation, it is still inefficient as data pages may be retrieved several times. In this section, we describe the join algorithm, the *superjoin* algorithm [23], incorporated into the NU-Prolog deductive database system, which is particularly suitable for top-down computation.

When answering the query ?–p(X,Y),q(Y,Z), the naive top-down approach finds a solution for p(X,Y) and then, using the binding generated for Y, finds solutions for q(Y,Z). This process, which is repeated for all solutions of p(X,Y), requires multiple re-scannings of the q relation and the possible re-reading of the data pages where the q facts are stored.

The *superjoin* algorithm organises access to the facts so as to minimise repeated retrieval of data pages. Unlike other join algorithms, this algorithm does not require a separate partitioning phase. Instead, it makes use of the clustering properties of the *dsimc* indexing scheme. Conceptually, each sub-goal q in the conjunction is evaluated as a number of sub-queries, where each sub-query $q(\overline{t})$ accesses a disjoint region of the external database containing the q facts. There is one such region for each hash value of the join variable, where we consider only that part of the hash value which is used in the index of both relations. Since these regions are much smaller than the total database, they can usually be buffered entirely in memory, so that no data page is read more than once during the processing of the join.

Superjoin fits naturally into Prolog's top-down computation rule and permits efficient processing of multi-way joins, where the join conjuncts may be interspersed with arbitrary Prolog goals. It can also be used effectively with the other computational methods discussed in the next section. In the context of a top-down computation, one disadvantage of *superjoin* is that it may require large amounts of buffer space when processing recursive rules.

6. Recursive Query Optimisation

Bottom-up computations deal with many tuples at a time and can naturally employ the existing optimisation techniques developed for relational databases. As a consequence, bottom-up computation is the focus of much research into deductive databases. We are developing a mechanism whereby the answers to database queries can be computed bottom-up within the framework of a general top-down computation.

Bottom-up methods for answering a query to a database are essentially refinements of the following scheme. A set of ground atoms S, initialised with the database facts, is constructed by repeatedly *applying* the rules to S until no new atoms are generated. A rule is applied to S by adding the head of a ground instance of the rule to S whenever all body literals of this instance are in S. The answers to the query are those instances of the query in the final set S. The scheme is guaranteed to terminate in the absence of function symbols because the number of ground atoms that can be constructed is finite.

The major drawback of naive bottom-up computation is the number of tuples which may be unnecessarily generated. Many tuples are *redundant*, that is, generated more than once. The differential approach suggested in [2,5,6] is one refinement to limit the number of such tuples. Using this approach, each tuple generated during an iteration is constructed in terms of at least one tuple generated during the previous iteration. Therefore, a set of tuples will not be used to repeatedly generate the same tuples at each iteration. In [2], the construction of the solution set is is expressed in terms of the iterative solution of a system of equations, where the equations are assignment statements which define how the tuples generated in one iteration are used to generate tuples in later iterations. In [3], the differential method is generalised for non-linear recursive databases.

Many tuples are *irrelevant*, that is, play no part in answering the query. We do not need to generate these tuples even once, and so we consider only relevant rules. *Relevant rules* are those whose head predicate is the query predicate, and predicates called by the query predicate, directly or indirectly. However, not all tuples generated by relevant rules help us to answer a given query. The standard bottom-up method generates many irrelevant tuples because it does not make use of ground terms in the query or variable bindings in the derived literals in the same goal-driven way that a top-down computation does.

There are two basic approaches toward limiting the generation of these irrelevant tuples. The first approach seeks to modify the standard bottom-up computation and achieve a run-time oriented solution. The second approach seeks to perform a compile-time transformation of the original database into an equivalent form which enables a standard bottom-up computation to focus on relevant tuples. Two examples of this approach are magic sets and counting sets. Our research [4] has investigated the use of *magic set* transformations on function-free databases that include negation in the body of rules.

There are two properties that such databases should have in order to answer queries both soundly and efficiently. The class of *stratified* databases were defined to make the model theory manageable when negative body literals are included in the database, by disallowing certain combinations of recursion and negation such as $p :- \neg p$. For this class, there is a natural way of selecting a particular minimal model as the intended one. Since a semantics has been defined for this class of databases, we require that the database be stratified. *Allowedness* [7, 14] is a syntactic characterisation of *domain independence* [25]; domain independence ensures that the semantics of a database (given by its set of positive logical consequences) does not change when new constants are added to the language [12]. Unlike other magic set algorithms, our algorithm preserves both stratification and allowedness. We have defined an extended notion of allowedness [4] and shown that, by using *allowed sideways information passing strategies*, an allowed database remains so after the magic transformation. In order to solve the unstratification problem, we have developed a new system which employs a labelling algorithm and a new magic set algorithm which generates a system of iterative equations whose solution retains the efficiency of *magic sets* for such databases.

7. Transaction Processing

For deductive database systems to be of practical use, their facilities for updating the database must ensure consistency in a multi-user environment. The standard Prolog primitives for database update (assert and retract) are not adequate. These primitives do not lend themselves to implementation in a multi-user environment. A simple modification to Prolog would be to call assert and retract between primitives which specify the start and end of a transaction. Concurrency control and integrity constraint checking could also be incorporated into this system. Problems with this approach include the undefined semantics of assert and retract [26], and inefficient integrity constraint checking.

In [16], we proposed a transaction primitive for updating database relations consisting of ground facts. This primitive has a well defined semantics and ensures that sets of insertions and deletions are performed atomically. Conceptually, a *transaction* is a function which maps an old database state, plus some input, to a new database state. A simple way to specify this is with a definition of the new database state in logic. However, for the current generation of implementations a lower level primitive is necessary for efficiency. Database state transitions are implemented by inserting and deleting facts into/from the database. We specify a transaction by defining two sets – the facts to be deleted (Del) and the facts to be inserted (Ins). We can define the new database state in terms of the old state by $New\_db = (Old\_db - Del) \cup Ins$.

Like the new database state, the insert and delete sets could be defined separately in logic. For example, suppose there is a single predicate p; for each fact in the database, we want to increment its argument, if greater than 20, by some input Increment. The insert and delete sets could be defined as:

$\forall$ A p(A) $\in$ Del $\leftrightarrow$ p(A), A > 20.

$\forall$ A,B p(A) $\in$ Ins $\leftrightarrow$ p(B), B > 20, A is B+Increment.

A major advantage this has over assert and retract is that there is a well defined *declarative* semantics for the overall change to the database. However, the naive implementation would access the entire p predicate twice, and as before, optimizing such computations is not trivial. We propose to have a single formula which allows the programmer complete control over the execution but defines both Ins and Del. We retain well defined declarative semantics, although they can be more difficult to extract. However, our proposal gives full expressibility and programmer control.

Our transaction operator has the following syntax:

all Local_vars transaction Goal

Local_vars is a term containing variables local to Goal (the same as other universally quantified variables in NU-Prolog [15]). If there are no local variables, "all Local_vars" may be omitted. Goal is a NU-Prolog goal which may contain calls to the procedures dbInsert and dbDelete. As an example, the previous specification of a transaction can be trivially transformed into our primitive as follows:

```
all A.B transaction (p(A), A > 20, dbDelete(p(A)) ;

                     p(B), B > 20, A is B+Increment, dbInsert(p(A))).
```

A more efficient version, which only calls p once, is:

```
all A.B transaction (p(A), A > 20, B is A+Increment, dbDelete(p(A)), dbDelete(p(B))).
```

When transaction is called, all solutions to the goal are found. These correspond to the dbInserts and dbDeletes called on the successful branch of the computation. Each solution may cause several insertions and deletions. However, they are saved rather than done immediately. After all solutions have been found the transaction attempts to *commit*. If this is successful, the updates are made – first all the deletions then all the insertions. To ensure atomicity of updates, the commit phase must employ some form of concurrency control, such as optimistic concurrency control. During this phase, integrity constraints may also be checked using the simplification procedure of [13, 14].

8. Conclusion

The techniques used in the NU-Prolog deductive database system may be effectively applied to a wide range of databases. Superimposed codeword indexing schemes provide efficient access to large databases of Prolog terms. Many useful systems can be implemented using the simple top-down computational method in combination with the superjoin algorithm. For large databases containing recursion, bottom-up methods are preferable, especially when their efficiency has been enhanced by the differential approach and magic sets. To make the system useful in multi-user environments, we employ a clean, concurrent update facility based on transactions.

Acknowledgements

We wish to thank all the members of our research group, who have provided a stimulating and energetic research environment. In particular, we would like to thank: John Lloyd, Ron Sacks-Davis, Rodney Topor, Liz Sonenberg and Alan Kent. We must also thank Jeff Schultz, who implemented the majority of the NU-Prolog system; without his efforts the implementation of the database would have been considerably longer in coming to fruition. Thanks must also go to Ann Nicholson and Zoltan Somogyi for increasing the readability and cohesion of the final product by their careful reading of earlier drafts. We wish also to thank the Minister for Science, Barry Jones, and Jean-Louis Lassez for initiating the Machine Intelligence Project at the University of Melbourne. This work was supported by the Commonwealth Department of Science, Pyramid Technology (Australia), the Australian Research Grants Commission, and the Commonwealth Scientific and Industrial Research Organisation.

References

1. I. Balbin, and K. Lecot, *Logic programming: a classified bibliography*, Wildgrass Books, Melbourne, Australia, 1985.
2. I. Balbin, and K. Ramamohanarao, A differential approach to query optimisation in recursive deductive databases, Technical Report 86/7, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.
3. I. Balbin, and K. Ramamohanarao, A generalisation of the differential approach to recursive query evaluation (1987).
4. I. Balbin, G. S. Port, and K. Rao, Magic set computations for stratified databases, Technical Report 87/3, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1987.
5. F. Bancilhon, Naive evaluation of recursively defined relations, *Proceedings of the Islamadora Conference on Database and AI*, 1985.
6. R. Bayer, Query evaluation and recursion in deductive database systems, Technical Report, Technische Universita't Mu'nchen-18503, Institut fuer Informatik, Technische Universitaet Muenchen, February 1985.
7. K. L. Clark, Negation as failure, in: H. Gallaire, and J. Minker (eds.), *Logic and data bases*, Plenum Press, 293-322, 1978.
8. R. S. Fabry, Capability-based addressing, *Communications of the ACM*, 17(7):403-412 (July 1974).
9. H. Gallaire, J. Minker, and J. Nicolas, Logic and databases: a deductive approach, *ACM Computing Surveys*, 16(2):153-185 (June 1984).
10. J. W. Lloyd, and K. Ramamohanarao, Partial-match retrieval for dynamic files:150-168 (1982).
11. J. W. Lloyd, and R. W. Topor, Making Prolog more expressive, *Journal of Logic Programming*, 1(3):225-240 (October 1984).
12. J. W. Lloyd, *Foundations of logic programming*, Springer Verlag, New York, 1984.
13. J. W. Lloyd, E. A. Sonenberg, and R. W. Topor, Integrity constraint checking in stratified databases, Technical Report 86/5, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.
14. J. W. Lloyd, and R. W. Topor, A basis for deductive database systems II, *Journal of Logic Programming*, 3(1):55-67 (April 1986).
15. L. Naish, Negation and quantifiers in NU-Prolog, *Proceedings of the Third International Conference on Logic Programming*, London, England, July 1986, pp. 624-634.
16. L. Naish, J. A. Thom, and K. Ramamohanarao, Concurrent database updates in Prolog, *Proceedings of the Fourth International Conference in Logic Programming*, Melbourne, Australia, May 1987.
17. K. Ramamohanarao, and J. W. Lloyd, Dynamic hashing schemes, The Computer Journal, November 1982.
18. K. Ramamohanarao, J. W. Lloyd, and J. A. Thom, Partial-match retrieval using hashing and descriptors, *ACM Transactions on Database Systems*, 8(4):552-576 (December 1983).
19. K. Ramamohanarao, and R. Sacks-Davis, Recursive linear hashing, *ACM Transactions on Database Systems*, 9(3):369-391 (September 1984).
20. K. Ramamohanarao, and R. Sacks-Davis, Partial-match retrieval using recursive linear hashing:477-484 (1985).
21. K. Ramamohanarao, and J. Shepherd, A superimposed codeword indexing scheme for very large Prolog databases, *Proceedings of the Third International Conference on Logic Programming*, London, August 1986, pp. 569-576.
22. R. Sacks-Davis, and K. Ramamohanarao, A two level superimposed coding scheme for partial match retrieval, *Information Systems*, 8(4):273-280 (1983).
23. J. A. Thom, L. Naish, and K. Ramamohanarao, A superjoin algorithm for deductive databases, *Preprints of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, D.C., August 1986, pp. 118-135.
24. J. A. Thom, and J. Zobel (eds.), NU-Prolog reference manual, version 1.0, Technical Report 86/10, Department of Computer Science, University of Melbourne, 1986.
25. R. W. Topor, Domain independent formulas and databases, Technical Report 86/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.
26. D. S. Warren, Database updates in pure Prolog, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, pp. 244-253.
27. J. Zobel, and K. Ramamohanarao, Accessing existing databases from Prolog, Technical Report 86/17, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

The Advanced Database Environment of the KIWI System

D. Saccà
Università della Calabria, Rende, Italy

M. Dispinzeri, A. Mecchia, C. Pizzuti
CRAI, Rende, Italy

C. Del Gracco, P. Naggar
ENIDATA, Rome, Italy

1. Introduction

The aim of the ESPRIT project *Knowledge-based User-Friendly System for the Utilization of Information Bases* is to design and to implement a prototype system (the *KIWI System*). This system allows the user to develop knowledge-based applications that make use of data from a number of external databases [S2]. Such applications are written in a knowledge representation language (called OOPS) that is basically object-oriented but enriched with other paradigms, particularly, logic programming. The OOPS language is supported by a database environment which is responsible for managing the knowledge base of the KIWI system as well as for retrieving and inferring facts from the knowledge base and external databases. External databases are existing operational databases, managed by a commercial DBMS, that are connected to the KIWI system through a suitable interface that runs as an application program for the DBMS. The knowledge base collects all pieces of information that are defined by the users of the KIWI System (note that the knowledge base can be updated whereas external databases are not).

The KIWI system consists of the following three layers:

a) *User Interface* (**UI**) that assists the users in the interaction with the KIWI system and helps him in identifying the needed information, retrieving it from the knowledge base and the available databases, and presenting the answer in a user-friendly way;

b) *Knowledge Handler* (**KH**) that implements all features of the knowledge representation language OOPS except those concerned with managing facts;

c) *Advanced Database Environment* (**ADE**) that receives requests from the KH for handling facts (both updating facts in the knowledge base and retrieving/inferring facts from the knowledge base and external databases).

The aim of this paper is to describe the design of the *ADE* layer that is currently under prototype implementation. *ADE* is based on the combination of logic programming and relational databases; the logic programming language supported by *ADE* is a simple extension of DATALOG (i.e., logic programming without function symbols) where base predicates correspond to objects rather than simple relational tuples. Actually, an object is vertically partitioned, thus it is represented as multiple tuples that are spread amongst several relations (a surrogate in every tuple is used to recompose the object).

To implement DATALOG programs, *ADE* uses a Prolog system coupled with relational databases for handling facts of the knowledge base and for retrieving additional facts from external databases. However, since DATALOG is based on the pure semantics of definite Horn clause queries (minimum model and fixpoint computation) that is not realized by the particular execution model of Prolog (SLD resolution with leftmost goal expansion), *ADE* modifies DATALOG programs into Prolog programs where the fixpoint computation is enforced. In addition, while rewriting a logic program, *ADE* selects a safe and efficient implementation, thus the programmer is relieved from the burden of ordering rules and goals; furthermore, if the underlying database is cyclic, a direct Prolog execution does not terminate at all, no matter is the chosen ordering. Following a "database approach" [U2], we compile the intensional information into efficient code that is applied at run time on the extensional database [BMSU, SZ1, SZ2, SZ3, BR]. The rational of compiling DATALOG programs into Prolog programs

where a bottom-up execution is enforced is three-fold:

a) The usage of Prolog allows to implement fast prototypes that can efficiently simulate relational algebra operations and fixpoint computation. The first experimentations on fact bases of medium size (around 500 facts) show that our compiled queries run much faster (up to 18 times) than their direct Prolog execution.

b) The user is given the possibility to pose the same query directly to Prolog or to have it compiled and executed in the same environment. Present systems combining Prolog and relational database system can be very much improved by this approach since they could provide an effective environment to support bottom up processing while continuing to offer the classical services of Prolog.

c) The output of a query compilation is a Prolog program whose main concern is to handle facts (assert or retrieve); if the Prolog system is extended to efficiently manage facts in both central and secondary memory, then our approach can be considered as a first short-term step toward the next generation database systems based on logics.

The compilation method used by *ADE* is the *Minimagic Method* that was first introduced in [SZ2] and later extended in [BR] and renamed *Supplementary Magic Set Method.* The Minimagic method improves on the Magic Set method [BMSU,SZ1] by avoiding duplicate computations. *ADE* uses an extension of the method based on dividing a logic query into subqueries that are independently computed [S1] rather than having a unique fixpoint computation as in [BR]. This method is used for compiling general logic program (thus with nested and mutual recursion and negation) and for propagating bindings whenever it is possible. In addition, safety issues are taken into account by carefully ordering goals in the rules.

The paper is organized as follows. In Section 2 we give an overview of the architecture of *ADE* and in Section 3 we present the compilation techniques and discuss other aspects of the implementation and further work.

2. Architecture of ADE

ADE is the software layer which allows the *KH* to deal with knowledge, both preexisting external databases and ad-hoc information defined by the user of the KIWI system (*Knowledge Base - KB*). The actual location of knowledge (i.e., whether in the *KB* or in external databases) is completely transparent to the *KH*. In fact, *ADE* has to properly dispatch any *KH* request, perform all the actions involved and send back a reply which can be either an outcome message or a set of objects in the answer. Furthermore, the *KH* world consists of objects, while the *KB* stores objects as database relations. Thus, *ADE* is also responsible for mapping the object system into a tuple system.

An object consists of a name (or surrogate) and of its relationships with other objects; it can be an *instance* of other objects; besides, it may have instances and, therefore, be a class; it may be a subclass of another object (*ISA* relationship), thus its instances are also instances of the other object; finally, an object may have properties (attributes) and values for those attributes. An object may contain the description of attributes for its instances.

The *KB* describes all objects using predefined relations such as:

- *OBJECT* (*Obj_name*), declaring all objects;
- *ISA* (*Subclass*,*Superclass*), defining *ISA* hierarchies between objects;
- *INSTANCE* (*Instance*,*Class*), defining instances;
- *OWN_ATTRIBUTE* (*Obj_name*,*Attr_name*,*Domain*), defining attributes for objects (the domain of an attribute is the set of instances of a class);
- *INSTANCE_ATTRIBUTE* (*Class*,*Attr_name*,*Domain*), defining attributes for all the instances of an object;
- *ATTRIBUTE_VALUE* (*Obj_name*,*Attr_name*,*Value*), assigning values to object attributes (such values are instances of attribute domains).

Obviously, the *KB* contains a number of implicit declarations for objects such as integer, strings and so on. In addition, it is possible to declare external classes, thus objects whose instances are preexisting tuples stored in external relational databases; in this case, additional information on such databases is to be supplied in order to let *ADE* retrieve such tuples.

The above relations represent the core of the *KB*; however, additional relationships among objects can be inferred using the following logic programming rules (the predicates isa^+ and $instance^+$ define the *ISA* and *INSTANCE* relationships while *isa* and *instance* correspond to the relations *ISA* and *INSTANCE*):

$isa^+(X,Y) :- isa(X,Y).$

$isa^+(X,Y) :- isa(X,Z), isa^+(Z,Y).$

$instance^+(X,Y) :- instance(X,Y).$

$instance^+(X,Y) :- instance(X,Z), isa^+(Z,Y).$

Moreover, the above two relationships allows objects to inherit further attributes in the following way:

$inst\_attribute^+(C,A,D) :- instance\_attribute(C,A,D).$

$inst\_attribute^+(C,A,D) :- isa^+(C,C'), instance\_attribute(C',A,D).$

$own\_attribute^+(O,A,D) :- own\_attribute(C,A,D).$

$own\_attribute^+(O,A,D) :- instance^+(O,C), inst\_attribute^+(C,A,D).$

It is easy to see that an object may have an attribute with several different domains; however, we require that the intersection of such domains is not empty.

In addition to the above relations, the *KB* contains a relation for every class object (called *database relations*). The attributes of a database relation are all the instance attributes of the object and the tuples are lists of values for those attributes, one for each instance of the object. As an example, consider two objects, *EMPLOYEE* and *PERSON*, such that *EMPLOYEE* is a subclass of *PERSON*. An instance of *EMPLOYEE* will be recorded as two tuples, one in the relation *EMPLOYEE* and the other in the relation *PERSON*. Surrogates in both relations allow to join the two tuples. It is worth noting that the logic programming language of *ADE* allows to refer in the *employee* predicate to attributes that are actually stored in the relation *PERSON*; *ADE* will eventually replace the predicate by a suitable conjunction of predicates corresponding to database relations.

The requests to *ADE* from the *KH* may be classified into:

- manipulation of objects describing the user knowledge (creating, updating, and deleting objects);
- submitting queries involving objects of the KB as well as relations of external databases (queries are expressed in terms of DATALOG programs).

If the *KH* sends a manipulation request, *ADE* must first check whether such a request could invalidate the coherence of the global objects system. Then, if the manipulation is acceptable, the *ADE* transforms the object-oriented request into a tuple-oriented one; therefore, a single object manipulation request may actually correspond to updating several tuples and may even activate other object manipulation requests. The relations of the *KB* are stored in a relational database under the exclusive control of *ADE*. Once the requested manipulation has been performed, *ADE* notifies the *KH* of the occurred knowledge modification.

Answering a query requires i) to replace object-oriented base predicates by tuple-oriented ones in which every base predicate in the logic program eventually corresponds to a relation (in the *KB* or in some external database), ii) to compile the logic program into a Prolog program using the Minimagic method and iii) to execute the program in a Prolog environment where a bottom-up processing is enforced and facts are retrieved from relational databases.

The architecture of *ADE* consists of two modules: *KNOW-ADE*, which is responsible for interfacing the *KH* layer and for mapping its requests into tuple oriented operations, and *ADE-MACHINE*, which is

responsible i) for managing the knowledge base, stored as a relational database, ii) for providing efficient implementation of logic queries and iii) for interfacing with external databases.

3. Compilation Techniques

We next present the approach used in *ADE* to implement DATALOG programs. We assume that "object-oriented" base predicates have been already replaced with base predicates corresponding to databases relations.

3.1. Logic Queries

We are given a number of relational databases [U1] $DB_1, \ldots, DB_k$ and a DATALOG program LP. Every predicate in LP can be i) a *comparison* predicate, ii) a *base* predicate corresponding to a database relation in $DB_1, \ldots, DB_k$, or iii) a *derived* predicate that is defined by LP. Notice that only derived predicates occur in the heads of rules.

Mutually recursive predicates cannot be solved independently from each other. We assume that LP is *stratified*, i.e., there exists no rule such that two mutually recursive predicates occurs in the head and in the body of the rule and the predicate in the body is negated [ABW, BNRST].

A program LP is composed by a number of subprograms, where every subprogram consists of all rules whose head predicates are mutually recursive. As an example consider the following DATALOG program:

r_0: $p(X,Y) :- b(X,Z), p(Z,W), c(W,U), U>X, p(U,V), not(b(V,Z)), d(V,Y).$

r_1: $p(X,X).$

r_2: $d(X,Y) :- b(X,Z), c(Z,Y).$

r_3: $c(X,Y) :- e(X,Z), f(Z,Y).$

r_4: $c(X,Y) :- b(X,Y).$

r_5: $f(X,Y) :- b(W,X), c(Y,Z).$

We have that predicates with symbols e or b are base predicates, whereas those with symbols p, d, f or c are derived predicates. The above program is composed by three subprograms: the first one with rules r_0 and r_1, the second one with rule rule r_2, and the third one with rules r_3, r_4 and r_5.

Given a derived predicate symbol p in LP, we denote by LP^p the subprogram of LP where the symbol p occurs in the head of some rule. In the example above, the first subprogram is denoted by LP^p, the second one by LP^d and the third one by LP^c or LP^f.

A *logic query* is a pair $Q=<G,LP>$, where G (*query goal*) is a derived predicate whose symbol occurs in the DATALOG program LP. The *answers* of a query Q are all facts that are in the stratified minimal model of LP and unify with the query goal G. (In stratified programs such a model exists and is unique [ABW]; if there are no negated predicates then this model coincides with the usual minimum model of Horn clause logic programs).

Given a query $<G,LP>$, where g is the symbol of the predicate G, the strategy used to answer the query is the following:

a) if the subprogram LP^g does not contain any derived predicate defined by another subprogram, then compute the fixpoint of LP^g and select all facts that unify with G;

b) if LP^g contains some predicate defined by another subprogram (say the predicate P), then fire the query $<P,LP^p>$ while computing the fixpoint of LP^g.

The above strategy is based on a query/subquery approach [V]. It is pointed out that, whenever it is possible, the fixpoint computation of every subprogram is done in two steps; in the first step, bindings are propagated to restrict the actual answer computation of the second step.

As an example consider the query $<p(a,Y),LP>$ on a database composed by the relations B and E (corresponding to the predicates in LP with symbol b and e, respectively), where LP is the program shown above. The query calls for the resolution of the subprogram LP^p; this subprogram, in turn, fires the subqueries $<c(W,U),LP^c>$ and $<d(V,Y),LP^d>$, where variables in the query goals are possibly replaced by constants. Finally, the resolution of the subprogram LP^d fires the subquery $<c(Z,Y),LP^c>$, again with suitable bindings.

3.2. Binding Propagation and Safety in a rule

Consider a recursive rule r in LP, say

$$r:\ P :-\ \mathbf{E}, \mathbf{Q}.$$

where P is the head predicate, **E** is a conjunction of database predicates, comparison predicates and derived predicates that are not mutually recursive with P, and **Q** is a conjunction of all derived predicates that are mutually recursive with P. Since r is recursive, **Q** is not empty. Rules r_0, r_3 and r_5 in the program shown in Section 3.1 are recursive.

Consider now a set of arguments of P denoted by a set S of argument positions (such arguments can be thought of as bound arguments). We denote the variables occurring in the above arguments by X_S. We are interested in recognizing whether the binding on the S-arguments can be passed to all recursive predicates in **Q**. For instance, take the following rule

$$r_0:\ p(X,Y) :-\ b(X,Z), p(Z,W), c(W,U), U>X, p(U,V), not(b(V,Z)), d(V,Y).$$

and assume that the first argument of the head predicate $p(X,Y)$ is bound, thus $S = \{1\}$. This binding is passed to the variable Z by the predicate $b(X,Z)$; so the predicate $p(Z,W)$ receives a binding and, in turn, binds the variable W. Moreover, the predicate $c(W,U)$ passes the binding to the variable U so also the predicate $p(U,V)$ is bound. Since there are no other predicates that are mutually recursive with the head predicate, we conclude that the rule propagates the binding $S = \{1\}$. Note that comparison predicates or negated predicates cannot be considered in the analysis of binding propagation since they must have all arguments bound before they can be safely solved.

More formally, we say that a recursive rule r *propagates the binding* S if if it is possible to order all predicates in **E**, **Q** as $R_1, \ldots, R_m$, such that

a) for each predicate R_i such that R_i is from **Q** or R_i is a non-comparison, non-negated predicate from **E** for which there exists a predicate R_j from **Q** with $j>i$, at least one variable occurring in R_i also occurs in X_S or in $R_1, \ldots, R_{i-1}$;

b) for each comparison or negated predicate R_i, all variables occurring in R_i also occur in X_S or in $R_1, \ldots, R_{i-1}$.

It is easy to see that also the rule

$$r_3:\ c(X,Y) :-\ e(X,Z), f(Z,Y).$$

propagate the binding $S = \{1\}$, whereas the rule

$$r_5:\ f(X,Y) :-\ b(W,X), c(Y,Z).$$

does not (but it is does propagate the binding $S = \{2\}$).

We call the rule $P :- R_1, \ldots, R_m$ the S-version of r. Rules r_0 and r_3 are already in the $\{1\}$-version; on the other side, the $\{2\}$-version of r_5 is

$$f(X,Y) :-\ c(Y,Z), b(W,X).$$

Suppose now that **Q** may be empty, thus the rule r can be also non-recursive. The rule r is *safe w.r.t.* a (possibly empty) binding S if it is possible to order all predicates in **E**, **Q** as $R_1, \ldots, R_m$ (*safe ordering*), such that

a) for each comparison or negated predicate R_i, all variables occurring in R_i also occur in X_S or in $R_1, \ldots, R_{i-1}$;

b) all variables occurring in the unbound arguments of P (i.e., those not denoted by S) appear either in X_S or in the body of r.

Note that the body of the S-version of a rule that propagates the binding S is safely ordered.

All the rules in the program shown in Section 3.1 but r_1 are safe w.r.t. $\emptyset$ and, then, w.r.t. any binding. On the other hand, the rule r_1 is safe w.r.t. $S = \{1\}$ or $S = \{2\}$.

3.3. Implementation of Subqueries.

Let $Q^H = \langle H, LP^h \rangle$ be a subquery and let I denote the bound arguments of the subquery goal H (I may be empty) and h be the predicate symbol of H. The implementation of a subquery is a Prolog program that enforces a bottom-up execution of it. The subquery Q has associated the predicate *query.h.I* with arity zero, that controls the bottom-up computation. It is stressed that the same compilation can be used for various subqueries that only differ in the actual binding but not in the binding pattern (i.e., the index set I).

Unbound Subquery

We have $I = \emptyset$. Let

r: P :- **E**.

be a (recursive or non recursive) rule in LP^h, where $\mathbf{E} = E_1, \ldots, E_k$ is safely ordered w.r.t. $\emptyset$. Then if r is non-recursive, the rule is modified into

query.h.$\emptyset$:- **E**, *not*(P), *assert*(P), *fail*.

otherwise the rule is rewritten as

query.h.$\emptyset$:- **E**, *not*(P), *assert*(P), *query.h.*$\emptyset$.

Consider the subquery $\langle c(W,U), LP^c \rangle$ with binding $I = \emptyset$, where LP^c consists of the rules r_3, r_4 and r_5. This subprogram is rewritten as follows

query.c.$\emptyset$:- $e(X,Z)$, $f(Z,Y)$, *not*($c(X,Y)$), *assert*($c(X,Y)$), *query.c.*$\emptyset$.

query.c.$\emptyset$:- $b(X,Y)$, *not*($c(X,Y)$), *assert*($c(X,Y)$), *fail*.

query.c.$\emptyset$:- $b(W,X)$, $c(Y,Z)$, *not*($f(X,Y)$), *assert*($f(X,Y)$), *query.c.*$\emptyset$.

Bound Subquery

We have that I is not empty. In addition, we require that every recursive rule in LP^h propagates the binding I as well as other possible bindings generated by I. For example, the subquery $\langle c(w,U), LP^c \rangle$ with binding $I = \{1\}$ is not bound since the rule

r_5: $f(X,Y)$:- $b(W,X)$, $c(Y,Z)$.

does not propagate the binding $\{1\}$. Therefore, this subquery is implemented as an unbound subquery. On the other hand, the subqueries $\langle d(v,Y), LP^d \rangle$ and $\langle p(x,Y), LP^p \rangle$, both with $I = \{1\}$, are bound.

In the implementation of a non-recursive rule, we make use of additional predicates corresponding to magic sets. For instance, the rule of the subprogram LP^d is rewritten as

*query.d.*1 :- *magic.d.*1(X), $b(X,Y)$, $c(Z,Y)$, *not*($d(X,Y)$), *assert*($d(X,Y)$), *fail*.

However, since the predicate $c(Z,Y)$ is derived, we need to fire a subquery; therefore, the rule becomes

*query.d.*1 :- *magic.d.*1(X),$b(X,Y)$, *not*(*query.c.*$\emptyset$), $c(Z,Y)$, *not*($d(X,Y)$), *assert*($d(X,Y)$), *fail*.

In the implementation of a recursive rule, we make use of other predicates, called supplementary magic set predicates; not only the rule is modified but also a number of rules are added to propagate bindings using magic set and supplementary magic set predicates. To give an intuition of the method, we next present the result of the compilation of $<p(x,Y),LP^p>$:

$$query.p.1 :- magic.p.1(X), b(X,Z), not(supmagic.r_0.1(X,Z)), \quad (1)$$
$$assert(supmagic.r_0.1(X,Z)), assert(magic.p.1(Z)), query.p.1.$$

$$query.p.1 :- supmagic.r_0.1(X,Z), p(Z,W), not(query.c.\emptyset), c(W,U), U>X, \quad (2)$$
$$not(supmagic.r_0.2(X,Z,U)), assert(supmagic.r_0.2(X,Z,U)),$$
$$assert(magic.p.1(U)), query.p.1.$$

$$query.p.1 :- supmagic.r_0.2(X,Z,U), p(U,V), not(b(V,Z), assert(magic.d.1(U)), \quad (3)$$
$$not(query.d.1), d(U,V), not(p(X,Y)), assert(p(X,Y)), query.p.1.$$

$$query.p.1 :- magic.p.1(X), not(p(X,X)), assert(p(X,X)), fail. \quad (4)$$

Rule (4) derives from the non-recursive rule r_1 whereas Rules (1)-(3) derive from the recursive rule r_0. Rules (1) and (2) propagate the bindings and Rule (3) actually computes the result. Notice that two subqueries ($query.c.\emptyset$ and $query.d.1$) are fired by Rules (2) and (3), respectively. Since $query.d.1$ is bound, before firing the subquery we need to transfer the initial binding through the predicate $assert(magic.d.1(U))$.

Implementation of a Query

Given a query $Q=<G,LP>$ with symbol g and bound arguments a denoted by I (if I is empty then the query is unbound), Q is implemented as follows:

1) The logic program is tested for stratification (actually, only the subprogram LP^g and all subprograms defining derived predicates in LP^g are considered). If it is not an error is reported.

2) The subprogram defining G is analyzed to see whether it propagates bindings and it is safe. If it is not safe an error is reported; otherwise, the subprogram is modified as shown in the previous sections. Furthermore, all subqueries (and their subprograms) that are fired by the subprogram are listed for a subsequent analysis.

3) The previous step is repeated for every subquery involved. Note that if there is an unbound query on a subprogram then it does not make sense to compute bound queries on the same logic program; therefore, only the compilation for the unbound query is to be kept.

4) If no error is reported in the previous steps, the overall modified logic program is passed to the Prolog-Database Environment for the execution.

5) The query is fired by the following goal

   $? assert(magic.g.I(a)), not(query.g.I), G.$

   that will give the answers to Q. On the other hand, if I is empty, then the query is fired simply by

   $? not(query.g.\emptyset), G.$

   Given the query $<p(a,Y),LP_1>$, the goal of the compiled program is

   $? assert(magic.p.1(a)), not(query.p.1), p(a,y).$

We conclude by pointing out that the Prolog programs generated by the *ADE* are more complex than the ones we have shown before since we need a better implementation of fixpoint computation by Prolog recursion. More details can be found in [S1] and in the design reports of *ADE*. Current work focuses on better interfacing to databases; in particular, we are analyzing the problem of efficiently handling database facts in a Prolog

environment. Finally, we are implementing other methods for query compilation, in particular the *Magic Counting* method [SZ4].

ACKNOWLEDGEMENTS. The project P1117 *KIWI - Knowledge-based User-Friendly System for the Utilization of Information Bases* is being carried out within the framework of the ESPRIT Program of the Commission of European Communities by a consortium composed by CRAI (Rende, Italy), Dansk Datamatik Center (Lyngby, Denmark), ENIDATA (Rome, Italy), INRIA (Le Chesnay, France), PHILIPS (Eindhoven, The Netherlands), University of Antwerp UIA (Antwerp, Belgium), Università "La Sapienza" (Rome, Italy). D. Saccà, who is participating to the project as member of CRAI's team, wishes to thank Carlo Zaniolo for many inspiring discussions.

4. References

[ABW] Apt, K.R., Blair, H.A., Walker, A., "Towards a theory of declarative knowledge", *Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, D.C., August 1986, pp. 546-628.

[BMSU] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D, "Magic sets and other strange ways to implement logic programs", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986, pp. 1-15.

[BNRST] Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, O., Tsur, S., "Sets and negation in a logic database language (LDL1), *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987, pp. 21-37.

[BR] Beeri, C., and R. Ramakrishnan, "On the power of magic", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987, pp. 269-283.

[S1] Saccà, D., "Compiling logic queries into Prolog programs", unpublished manuscript, 1987.

[S2] Saccà, D., Vermeir, D., D'Atri, A., Liso, A., Pedersen, S.G., Snijders, J.J., Spyratos, N., "Description of the overall architecture of the KIWI System", in *ESPRIT '85: Status Report of Continuing Work, The Commission of European Communities (Eds)*, Elsevier Publishers B.V. (North-Holland), 1986, pp. 685-700.

[SZ1] Saccà, D., Zaniolo, C., "On the implementation of a simple class of logic queries for databases", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986, pp. 16-23.

[SZ2] Saccà, D., Zaniolo, C., "Implementation of recursive queries for a data language based on pure Horn clauses", *Proc. 4th Int. Conf. on Logic Programming*, Melbourne, 1987, pp. 104-135.

[SZ3] Saccà, D., Zaniolo, C., "The generalized counting method for recursive logic queries", *Proc. 1st International Conf. on Database Theory*, Rome, 1986, to appear in *TCS*.

[SZ4] Saccà, D., Zaniolo, C., "Magic counting methods", *Proc. ACM SIGMOD Conf.*, San Francisco, 1987, pp. 49-59.

[U1] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, Rockville, Md., 1982.

[U2] Ullman, J.D., "Implementation of logical query languages for databases", *TODS 10*, 3, 1985, pp. 289-321.

[V] Vieille, E., "Recursive axioms in deductive databases: the query/subquery approach", *Proc. 1st Int. Conf. on Expert Database Systems*, Charleston, S.C., 1986.

YAWN! (YET *ANOTHER* WINDOW ON NAIL!)†

Katherine Morris
Jeffrey F. Naughton‡
Yatin Saraiya
Jeffrey D. Ullman
Allen Van Gelder††

Stanford University

ABSTRACT

We present the important features of the NAIL! knowledge-base system. These include the "capture-rule" architecture for integrating different logic-evaluation strategies, the NAIL! strategy-selection algorithm, and the NAIL! intermediate code for coupling with a database back end.

I. WHAT MAKES NAIL! DIFFERENT?

The NAIL! (Not *Another* Implementation of Logic!) project at Stanford is, loosely speaking, a "knowledge-base system." Its goal is to process queries posed in a Prolog-like language, and to process them as efficiently as a database system could. The central technical issues concern optimization of logic programs and the way the system can determine the most appropriate way to process a given query.

As an aid to the study of query optimization, we rely on an SQL database system, running on an IBM PC/RT, to perform the usual database tasks, and we have concentrated on implementing the front end and optimization portions of the system. Evidently, we lose performance by this "loose coupling" of the logic program and the database, but we do not lose the ability to experiment with the design of the query processor, and to develop the techniques that are essential to the next generation of database/knowledge-base systems.

In this section we discuss some features of the NAIL! system that make it different from most other attempts to combine logical interfaces with the capabilities of a database system. These ideas include:

1. The interface between predicates ("adorned goals" and relations) that allow different evaluation strategies to apply to different pieces of a logic program.
2. The two-part "capture rule," separating the test for applicability of an evaluation strategy from the execution of that strategy.
3. The extensible architecture, allowing the system to expand its query-processing capability gracefully.

Adorned Goals and the Predicate Interface

The NAIL! system architecture follows the "capture rule" approach outlined in Ullman [1985]. A fundamental assumption of our approach is that

No single query-evaluation strategy is appropriate
for all logic programs

Thus, the capture-rule architecture provides the "glue" that allows us to break apart logic programs into small pieces, find the best algorithm for each piece, and combine the various algorithms

† Work supported by NSF grant IST-84-12791 and a grant of IBM Corp.

‡ Present address: Dept. of CS, Princeton University

†† Present address: Dept. of CIS, University of California, Santa Cruz

into a whole that answers the query efficiently. More specifically:

1. NAIL! considers each predicate individually, except for groups of mutually recursive predicates, which must be handled as a group. A predicate, together with its mutually recursive predicates, if any, is called a "strong component."
2. When we choose an evaluation strategy for each predicate or strong component, we must pay attention to the *adornment* of the predicate, that is, to the bound-free pattern of its arguments. In response to a single query, it is possible that we must evaluate the same predicate with several different adornments, and different strategies might be used for different adornments. We use superscripts b and f to indicate adornments. Thus, p^{bff} represents the predicate p, which is a three-place predicate, and indicates that we wish to find the relation for predicate p with the first argument bound (to a given finite set of values) and the other two arguments free. We shall later give several examples to illustrate why bound-free patterns are important.
3. The result produced by the system in response to an adorned predicate is the relation for that predicate, as defined by the logical rules and the database, with the selection operator applied to reflect any bindings of arguments in the adornment. For example, suppose we are given as an adorned subgoal p^{bbf} for the predicate $p(X, Y, Z)$, and we are given a relation R consisting of (X, Y) pairs that provides the binding for the first two arguments. Then NAIL! will produce a relation $P(X, Y, Z)$ consisting of all those tuples t in the relation naturally associated with predicate p (according to the "iterated fixed-point semantics" of logical rules—see Minker [1988], Ullman [1988]), such that the projection of t onto the first two components, is in R. In this computation, the relation R not only limits the size of the result, but the fact that it restricts X and Y to finite sets may be essential if we are to compute the relation P at all, as discussed in Example 1, below.
4. NAIL! selects, by a top-down process to be described, the appropriate adornments for subgoals. It then builds the answer to the query bottom-up, combining the relations constructed for the adorned subgoals in two ways.
   a) The natural join of the relations for the subgoals of rules is taken to obtain a single relation for the body of that rule.
   b) A predicate that is the head of several rules has its relation computed by taking a selection and projection of the relation computed for each rule, to produce a relation whose components correspond to the arguments of the head. We then take the union of the relations produced from each rule.

   We illustrate some of this process in Example 2, below.

Example 1: To see why adornments can be important, consider the rules

```
goodList([]).
goodList([H|T]) :- good(H) & goodList(T).
```

For those not familiar with the notation of Prolog, which is used by NAIL! (except for the ampersand where most Prologs use comma), the first rule says that "the empty list is a good list." The second rule says that "a list with head H and tail T is a good list if H is good and T is a good list." Presumably, *good* is a database predicate; that is, we are given a collection of good elements. The predicate *goodList* is defined by its rules.

A query $goodList(X)$, is interpreted by NAIL! to mean "find all those lists X such that $goodList(X)$ is true.† Such a query cannot be answered, because it is an infinite set, as long as the relation *good* is nonempty. That is to say, the adorned goal $goodList^f$ cannot be "captured" by any method whatsoever.

---

† Note that this interpretation, which corresponds to iterated fixed-point semantics is not what Prolog would assume. Rather, Prolog would search for good lists one-at-a-time. NAIL! takes the point of view that all solutions are requested because that is consistent with the way queries to databases are interpreted.

On the other hand, a query $goodList(a)$, where a is a list or a finite set of lists, can be answered; one has only to check that each element on a list is *good*, and return the subset of the given set of lists that consist only of *good* elements. That is, we say $goodList^b$ can be "captured" by an appropriate capture rule. □

Example 2: Consider the rule

```
p(X,Y) :- q(X,Z) & r(Z,Y).
```

Suppose we are given a set of X-values for p, say $\mathcal{X}_0$. We thus have the adorned goal p^{bf}. If Q and R are the relations for predicates q and r, then we want to produce†

$$\sigma_{X \text{ in } \mathcal{X}_0}(\pi_{X,Y}(Q(X,Z) \bowtie R(Z,Y))$$

One reasonable way to do so is to solve the adorned subgoal q^{bf}, using the same set $\mathcal{X}_0$ to provide the bindings for X, the first argument of q. Let us suppose that this subgoal can be solved; for example, q might be a database predicate, so Q, its relation, exists physically in the database.

The solution to subgoal q^{bf} with binding $\mathcal{X}_0$ is a relation $Q_0(X,Z)$. On our assumption that q is a database predicate, the solution is $Q_0 = \sigma_{X \text{ in } \mathcal{X}_0}(Q)$. We can now project this relation onto Z to get a set of constants $\mathcal{Z}_0$ that serves as a binding for the first argument of the subgoal $r(Z,Y)$. Thus, we may next attack the adorned subgoal r^{bf} with binding set $\mathcal{Z}_0$. Note how the process of *sideways information passing*, where the value(s) of Z in the first subgoal $q(X,Z)$ were passed to the occurrence of Z in the second subgoal $r(Z,Y)$ is essential for efficiency. For example, there might be an index on the first argument of the database relation for R, allowing the retrieval to take place without scanning the entire relation R.

Suppose that the adorned subgoal r^{bf} can be solved too, and let the resulting relation be $R_0(Z,Y)$. Then we can compute the relation for p with binding set $\mathcal{X}_0$ by the expression $Q_0(X,Z) \bowtie R_0(Z,Y)$. □

The Two Parts of a Capture Rule

What we call a "capture rule" is really a pair of algorithms. The first, called the *test*, is used when we plan the query. We are given an adorned goal; we examine its rules and the rules of any other predicates in the same strong component, and we decide whether a given capture rule applies.

Example 3: Suppose we are given the adorned goal $goodList^b$ in Example 1. It is easy to see that Prolog's top-down, or "backward-chaining" approach will work; successive calls to *goodList* are on shorter lists, and eventually the recursion stops. There are several algorithms (Ullman [1985], Naish [1986], Afrati et al. [1986]) that are sufficient conditions for this convergence to occur. Thus, the first part of a "top-down" capture rule could be one of these algorithms. □

The second part of the capture rule, which we call the *substantiation*, is the evaluator of the relation for the given subgoal. It is only called if the adorned subgoal that is the query, and all of its resulting subgoals, are successfully captured by (the test portion of) some capture rule known to the system.

For example, the test portion of the top-down capture rule alluded to in Example 3 might determine that $goodList^b$ could be captured, provided $good^b$ could be captured. Another capture rule, one for database predicates, would come into play and tell us that indeed, $good^b$ can be captured. The corresponding substantiation would be database lookup.

Sufficient conditions for convergence of the top-down expansion of goals are not new, and of course, no condition that is both necessary and sufficient exists because the problem is undecidable. What is new in our framework is the emphasis on efficiency, both of the substantiation algorithms—the time to evaluate a query is important when large amounts of data are involved—and of the test algorithms.

---

† We use the relational algebra notation of Ullman [1982]; σ is selection, π is projection, and $\bowtie$ is the natural join.

Work on correctness and termination of programs assumes that such proofs are an infrequent activity, so generality of tests is more important than their efficiency. In a knowledge-base system, decisions about applicability of a capture rule must be made many times per query, so we cannot afford slow tests. Further, it is not acceptable to send the system on a "wild goose chase," applying some substantiation algorithm that supposedly will succeed, but in fact goes into an infinite loop because it is not applicable to the logic to which is is applied. Thus:

We need to study both fast ways to evaluate queries and fast ways to tell whether a given strategy will work

Extensibility of the Capture-Rule System

NAIL! is designed so that capture rules for a variety of kinds of logic can be added as the system evolves. The clean interface—adorned goals and returned relations—makes this extensibility easy.

However, we should be aware that no finite set of capture rules will capture all the programs that could ever be written in the NAIL! language. Since all capture rule tests are sufficient conditions, we know that when the system finds a set of capture rules whose tests succeed on the query's goal and all its subgoals, then the substantiation portions of these capture rules will return the correct relation. We also know that if no capture rule can be found for one or more subgoals, then the system will terminate with a failure message, that it does not know how to evaluate the query.

Now there are some queries expressible in NAIL! that have no answer, as in Example 1, e.g. That is not a special problem of NAIL!; one can write a Pascal program with an infinite loop, which therefore gives no answer. More of a problem is that NAIL! may fail on programs where there is some evaluation algorithm known to the user, but not to the system. Only experimentation will tell us whether that is a serious problem. More likely, we can implement all of the common logic-evaluation strategies as capture rules, and thus appear as "smart" as the user when determining evaluation strategies.

II. THE NAIL! ARCHITECTURE

A rough outline of the NAIL! architecture is shown in Fig. 1. The input to the system consists of a logic program, written in the NAIL! source language, and a sequence of queries. The following processing steps take place.

1. The rules (logic program) are preprocessed into an internal form and stored.
2. Queries are passed to a strategy-selection subsystem, which attempts to find the best way to implement the query. Strategy selection makes use of stored facts about successful ways to answer queries of certain forms (the *strategy database*), and we add to this body of strategy facts any discoveries we make about what is or is not successful in capturing the current query or any subgoals it requires.
3. When strategy selection is successful, an ICODE intermediate language program is generated, representing the algorithm that will be used to solve the query.
4. The ICODE program is modified by a code optimizer.
5. The ICODE program is interpreted by a subsystem that executes operations on relations by calls to the SQL database system.

The NAIL! Language

The NAIL! language is essentially pure Prolog with the additional operators *not* (for negation) and *findall* (for set formation). In order that a sensible semantics can be placed on logical rules involving negation and set-formation, we require that rules be *stratified* with respect to the *not* and *findall* operators. That is, we cannot define a predicate recursively in terms of its own negation or in terms of sets constructed from itself. Morris, Ullman, and Van Gelder [1986] contains more detail on this language and its semantics.

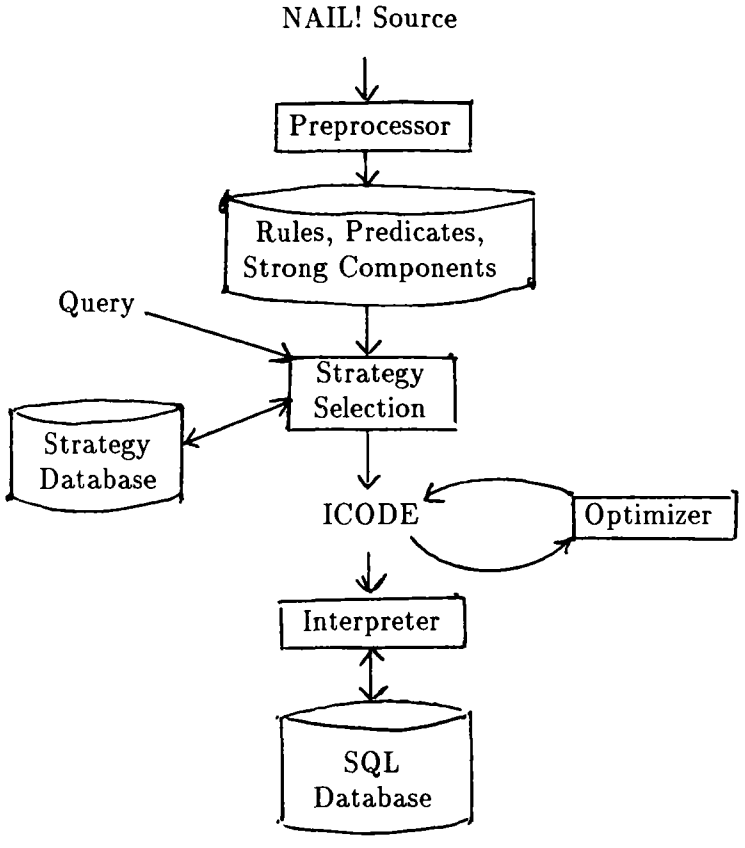


Fig. 1. The NAIL! architecture.

As was mentioned, the interpretation of logical rules is that of the iterated fixed-point. That is, the predicates are divided into strata, and if a predicate p depends on predicate q in a way that involves negation or set-formation, then q is (in principle) evaluated completely before computation of p is attempted; we can do so because of the "stratification" condition in the NAIL! source language; that is, q cannot depend upon p in this situation.

The relations in the NAIL! system are divided into two disjoint classes. The Extensional Data Base or EDB relations are stored in a traditional relational database system, as relations. The Intensional Data Base or IDB relations are defined in terms of other relations, by the rules of the NAIL! program.

The Rule Preprocessor

Some rewriting of the rules is necessary in order to isolate *not* and *findall* constructs, and to expand operators such as "or" found in the bodies of rules. The stored form of rules thus allows constructs of only three types:

1. Rules whose body (right hand side) is a conjunction of positive literals,
2. Rules whose body is a single negated literal, and
3. Rules whose body is a single *findall* subgoal.

Currently, only the first two of these types are implemented.

In addition, the preprocessor produces a *predicate graph*. This graph contains, for each rule in the NAIL! program, arcs from the predicate at the head (left side) of the rule to each predicate of the body.

Finally, the predicates are partitioned according to the SCC's (*strongly-connected components*)

of the predicate graph; doing so enables us to detect mutually recursive predicates. The SCC's are the elementary units about which we make decisions concerning the strategy to use to evaluate the relation associated with a predicate.

Example 4: Predicate *sg*, the familiar "same-generation" predicate, can be defined by two rules. The first says that any individual is his own cousin. The second says that X and Y are cousins at the same generation if they have parents who are cousins at the same generation.

```
(1) sg(X,X).
(2) sg(X,Y) :- par(X,W) & par(Y,Z) & sg(Z,W).
```

Note that we have purposely switched the order of the arguments in the recursive call to *sg*, using Z, W instead of the more typical W, Z. As *sg* is symmetric, the meaning of the predicate is not changed, but it leads to an instructive example later on.

A second predicate, *related*, is the transitive closure of *sg*:

```
(3) related(X,Y) :- sg(X,Y).
(4) related(X,Z) :- related(X,Y) & related(Y,Z).
```

Figure 2 shows the predicate graph for these rules. In this example, each predicate is in an SCC by itself, and there are two recursive predicates, `sg` and `related`. □

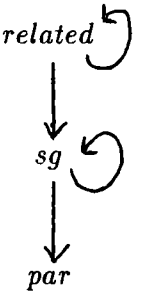


Fig. 2. Predicate graph for Example 4.

III. THE NAIL! INTERMEDIATE CODE

The NAIL! ICODE is essentially statements of relational algebra with flow of control, procedure calls, and stacks. A "manual," Ullman [1986], is available. Here, we shall only give an example of an ICODE program and mention some "features." The code in Fig. 3 implements the algorithm of Henschen and Naqvi [1984] on the same generation problem; that is, given one or more individuals (in a unary relation called *query*), it finds all the cousins of those individuals.

We shall introduce the ICODE language by annotating each of the lines of code.

1,2: The relations *par* and *query* are declared, and their attributes named, so NAIL! can communicate with the SQL database.

3: The ICODE variable *up* will be used as a stack, with successive levels holding unary relations that are sets of ancestors of the individuals in the relation *query*. We initialize *up* to be the empty stack.

4: Push the query set, or first generation, onto the stack *up*. Later, the parents of the individuals in *query* will occupy the second level, the grandparents the third level, and so on.

5: A Prolog rule is a legal ICODE statement. This rule asks us to take the join of the relation on top of the stack *up*† with the *par* relation. The result is a new set of ancestors, the parents of the individuals at the current top of stack.

† A stack name, by default, refers to the top of the stack. We could address other levels of the stack, e.g., *up*[−2] for the relation two levels below the top, *up*[5] for the fifth level from the bottom, and *up*[*i*] for the *i*th level, whatever integer-valued ICODE variable *i* is at the moment.

```
        % Henschen-Naqvi method, using stack
(1)         relation(par(child,parent))
(2)         relation(query(person))
(3)         makeEmptyStack(up)
(4)         push(query,up)
        label(l)
(5)         newup(X) :- up(Y), par(Y,X).
(6)         ifEmpty(m, newup)
(7)         push(newup, up)
(8)         goto(l)
        label(m)
(9)         assign(down, up)
        label(o)
(10)        pop(up)
(11)        ifEmptyStack(n, up)
(12)        newdown(X) :- par(X,Y), down(Y).
(13)        union(down, newdown, up)
(14)        goto(o)
        label(n)
(15)        print(down)
(16)        halt
```

Fig. 3. Henschen-Naqvi algorithm in ICODE.

6: If the new set of ancestors is empty, we break out of the loop; we have reached the top of the genealogy and can proceed to work down the generations, finding descendants of the individuals on the stack. We go to the statement numbered 9, which follows the target label, *m*.

7,8: Otherwise, we push the new generation onto the stack and repeat the loop of lines 5–8.

9: Relation-valued variable *down* holds the descendants of the individuals on the stack, at progressively lower generations. Initially, *down* holds the members of the highest generation.

10: Here, we begin a loop, one time around for each generation, back down to the original generation of the individuals of the set *query*. Each time around, we pop the stack.

11: If we have reached the last generation, then *down* is our answer. We go to line 15 to finish up.

12: Here, we have not yet reached the last generation. Compute the children of the current *down* set.

13: Add to *down* the individuals on the stack at that generation.†

14: Repeat the loop of lines 10–13.

15: Here, we have reached the last generation. Print the answer.

16: And halt. □

Additional ICODE Capabilities

The ICODE permits function symbols to appear in steps that are Prolog statements. The translation into SQL makes use of a permanent relation CONS that, in effect, stores "cons cells," allowing us to construct complex terms from tuples of relations. Each cons cell has its own "tuple ID," an attribute of the CONS relation. We arrange that there can never be two different cells representing

† It happens that each individual in the relation at the top of the *up* stack must necessarily be a member of *down* at this time. We included this statement both to illustrate the use of the *union* operation and because to do so is more faithful to the Henschen-Naqvi algorithm, where a more complicated basis rule than $sg(X, X)$ would require the union here.

the same term. Thus, tests for equality of terms are made simple when translated into SQL; they become tests for equality of ID's. Of course, insertion into CONS is thus made harder than it would otherwise be.

The ICODE also includes

1. Call and return statements.
2. Set difference.
3. Selection, projection, and join (which are special cases of Prolog statements, but use a more standard relational-algebra syntax).
4. Ordinary computation on integer-valued variables.

IV. DATA STRUCTURES USED IN STRATEGY SELECTION

Before considering the details of the strategy-selection algorithm that is the heart of NAIL!, let us consider in more detail the data structures that NAIL! uses to represent its information.

Rule/Goal Graphs

When we consider algorithms for answering queries, we have to make certain decisions about the way information is passed sideways in an SCC.

Example 5: Recall that in Example 2 we had the rule

```
p(X,Y) :- q(X,Z) & r(Z,Y).
```

and, presented with the adorned goal p^{bf}, we chose to work on q first, thus binding Z for the purposes of the second subgoal, and generating adorned subgoals q^{bf} and r^{bf}.

We also could have worked on the second subgoal first, passing Z-bindings from the second to the first subgoal, and yielding adorned subgoals r^{ff} and q^{bb}. Generally, we prefer to work on two subgoals that are partially bound, rather than one all of whose arguments are free, because binding even one argument tends to cut down markedly the amount of data generated. However, there may be some reason why it would be necessary to follow the second approach. For example, q^{bf} may be uncapturable by any means, although q^{bb} is capturable (if, say, the definition of q involves a recursion on the length of the list that is the second argument, as in Example 1). □

The data structure we use to represent these (perhaps tentative) decisions is called a *rule/goal graph* (RGG). The RGG is derived from the predicate graph for a SCC by:

1. Indicating *adornments* for goals.
2. Indicating the order in which the subgoals of a rule are to be processed, thus implying a direction for sideways information passing.

It is important to realize that one predicate in the SCC may be expanded to many occurrences in the RGG, each corresponding to a different adornment of that predicate.

Goal nodes in an RGG correspond to adorned goals. Rules are represented by a chain of *rule nodes*, the first of which represents the entire body; successive nodes represent progressively smaller subsets of the subgoals in the body. More precisely, a rule node representing set of subgoals S has a left child representing one of the subgoals s in S, and a right child representing $S - \{s\}$. If S is a singleton, the right child is omitted. The adornment of each rule node is of the form

(<bound variables> | <free variables>)

The set of bound variables becomes larger as we go down the chain for a given rule. At the top, only the variables bound by the head of the rule appear on the left of the bar. Suppose rule node n has left child g and right child m. Then a variable X is bound at m if either it is bound at n, or X appears in an argument of the subgoal g.

Example 6: Let us consider one possible RGG for the SCC of *sg* in Example 4, whose rules are repeated here:

r_1: $sg(X, X)$.
r_2: $sg(X, Y)$:- $par_1(X, W)$ & $par_2(Y, Z)$ & $sg(W, Z)$.

We have subscripted the two occurrences of the predicate *par* to distinguish them in the discussion below.

Let us start with adorned goal sg^{bf}, that is, a query like `sg(joe,Q)`, whose first argument is bound and whose second is free. The RGG, shown in Fig. 4, is based on the idea that we should evaluate the subgoals of rule (2) in the order that allows the maximum passing of bindings sideways. That is, we take the binding of X from the head of that rule, and use it to evaluate the first subgoal $par(X, W)$ with its first argument bound, i.e., we have the adorned goal par_1^{bf}. The binding of W produced by that evaluation is used in the third subgoal to produce an instance of the adorned goal sg^{fb}, and that provides a binding for Z, which may be used to bind the second argument in the second subgoal, $par(Y, Z)$. The expansion of r_2 thus concludes with par_2^{fb}.

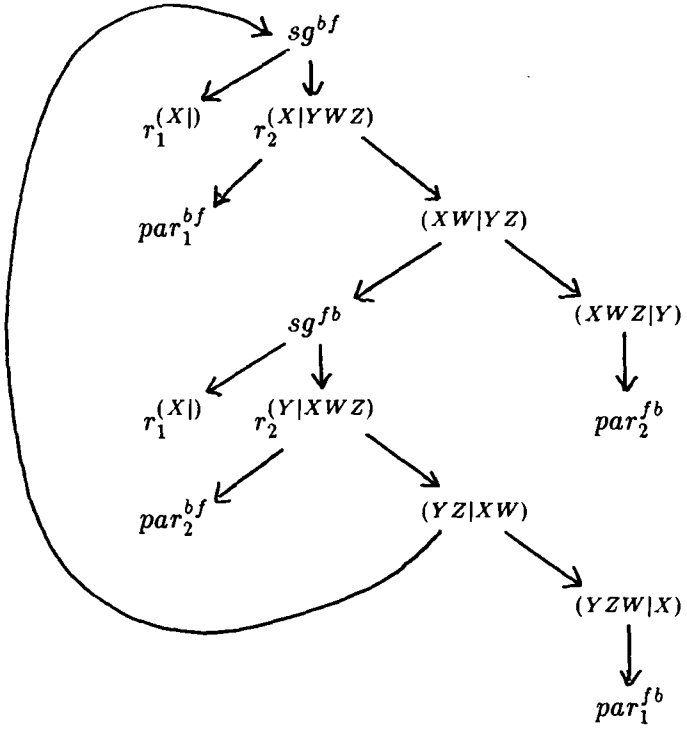


Fig. 4. A rule/goal graph for the *sg* predicate.

Now, to complete the RGG, we must consider the node for adorned goal sg^{fb}. We again must deal with rule (2), and here the most sensible choice is to take the binding on Y of the head, pass it to the first argument of the second subgoal, to get another instance of par^{bf}. That gives us a binding for Z, which lets us convert the third subgoal into an instance of sg^{bf}, which is already in the RGG. Finally, the binding for W produced by this subgoal is used to create another instance of par^{fb} out of the first subgoal. □

There are a few comments that are worth making about the rule/goal graph and Fig. 4 in particular.

1. The goal nodes of an RGG are divided into *internal* nodes, whose predicate belongs to the SCC of the goal being expanded, and *external* nodes, whose predicate is from a different SCC (which must be "lower," since there can be no cycles among the SCC's themselves). We never expand the external nodes, only the internal ones, while working on this SCC.
2. It helps to think of the relation for a goal node as the union of projections of the relations for its successor rule nodes. The actual process is slightly more complicated, since the heads of the various rules may have different variables, and there may be repeated variables or constants in heads, e.g. $p(X, a, X)$. However, the construction of a relation for a goal node can always be performed by the operations selection, projection, and union.
3. Likewise, the relation for a rule can be constructed by join operations from the relations for its successor goal nodes.
4. Points (2) and (3) may have the reader wondering whether this view makes any sense for RGG's with cycles, i.e., those with recursive predicates in the underlying SCC. We do not wish to imply that any evaluation algorithm can be applied to any RGG. There are some algorithms, such as Henchen-Naqvi, or the "magic-sets" algorithm of Bancilhon et al. [1986], that can be applied to the RGG of Fig. 4, and the strategy-selection algorithm of NAIL! can find these when presented with the adorned goal sg^{bf}.

Annotated Rule/Goal Graphs

Now, let us consider in more detail what happens to an adorned goal set up in response to a query such as `sg(joe,X)`. As we mentioned in Example 6, the first step is to create the adorned goal from the query, by determining which of the arguments are bound and which are free. In this case we have the adorned goal sg^{bf}.

The next stage is to attempt to *capture* the adorned goal. Informally, we can capture an adorned goal if we know how to compute its result relation from the database. Our knowledge about capturing goals is embodied in capture rules. The test portion of a capture rule must examine the SCC for the goal and decide whether the capture rule applies. Normally, the test calls an RGG constructor, which is given a set of "forbidden adornments" (described in the next section), which it may not use in the RGG. The test then reduces to deciding whether an RGG containing the adorned goal exists.†

If it succeeds, the capture rule will return an *annotated, adorned, rule-goal graph* (AARGGh). If it fails, we try another capture rule, should the system have appropriate, untried capture rules available.

The term "annotation" refers to information describing a strategy for query evaluation. We say a goal or rule is *annotated* if it has an associated strategy for capturing it. The AARGGh is a list of arcs from adorned goals in the SCC to *annotated, adorned rules* (AAR's), and from annotated, adorned rules to other adorned goals in the SCC. The rules in the AARGGh are annotated with the strategies used to solve them, and this strategy information recursively includes the AARGGh's for subgoals of the rule that are outside the SCC.

The Strategy Facts Database

There are two types of strategy information: *capturable* and *uncapturable* facts. Capturable facts are stored whenever we successfully capture an adorned goal; the information stored includes the AARGGh generated, the name of the capture rule used, and other information specific to the capture rule. Uncapturable data is stored when we are unable to capture an adorned goal; it contains the name of the capture rule and the adorned goal only. The strategy information is kept until the NAIL! rules are changed, or until more capture rules are added to the system. Clearly we cannot assume the same strategies will be selected after either of these occurrences.

---

† Other kinds of tests are possible, but not supported directly by the system.

A fundamental assumption we make is that if we can use a particular capture rule to capture an adorned goal p^{α}, then we can also use it to capture p^{β}, where β is any adornment that is *more bound* than α, by which we mean that wherever α has b, β does too, but β may have a b where α has f. We call this assumption the *bound-is-easier* assumption.

The strategy facts and the bound-is-easier assumption are used as follows. Suppose we have a query `tricky(a,Y,Z)`, and we have decided on a capture rule C to use. When we look up the strategy information for $tricky^{bff}$, we discover that $tricky^{bbf}$ is not capturable using C. Since this easier goal is not capturable, we know we cannot capture our goal using C. Similarly, if we find we can capture a more difficult goal, in this case $tricky^{fff}$, we know we can capture the goal using a selection on the entire relation for predicate *tricky*. That is, given a set S of bound values for the first argument of *tricky*, we can first find the relation T obtained by applying the capture rule C to the adorned goal $tricky^{fff}$. We then select those tuples of T whose first component is in S.

V. THE NAIL! STRATEGY SELECTION ALGORITHM

The heart of the NAIL! system is an algorithm to take an adorned goal and discover an algorithm whereby its relation can be computed efficiently. When designing the strategy-selection algorithm, we were faced with a problem that is familiar to those working in logic programming: the trade off between top-down and bottom-up processing. Bottom-up processing ("forward chaining") tends to produce more inferences per second, but there is a lack of "focus"; we prove facts that may or may not have any relevance to our goal. Top-down ("backward-chaining") processing is more focussed on the goal at hand, but tends to take longer per inference and may enter an infinite loop on rules for which bottom-up processing would converge.

Similarly, in NAIL!, we could process all adorned goals bottom-up, deriving the best strategy for each, starting with EDB predicates, then those predicates that depend only on EDB predicates, and so on. We expect that for typical sets of rules, only a small fraction of all adorned goals will ever need to be solved, so we decided to proceed top-down, solving the strategy for an adorned goal only if required to do so to answer a query. As we store all discovered strategies in the strategy database, we get most of the advantage of bottom-up processing as well, and we never have to recompute strategies.

Reducing the Strategy Search

Suppose we have an SCC, such as that of predicate p from Example 2, that has only the rule

```
p(X,Y) :- q(X,Z) & r(Z,Y).
```

Also suppose p^{bf} is the adorned goal. We attempt to apply a capture rule C to this simple case; perhaps C merely calls for evaluation of the relations for q and r, followed by a join and projection. In general, however, we do not know whether C will work for p, and we must explore the structure of the rule(s) for p and also determine whether any adorned subgoals that get generated can themselves be captured.

Worse, the particular set of adorned subgoals we generate depends on choices we make for p. In general, that set depends on the rule/goal graph we choose for p's strong component. In this simple case, we have only the choice of which subgoal to work on first. Recall from Example 5 that we have the choice of solving q^{bf} and r^{bf} or of solving q^{bb} and r^{ff}; either may be possible while the other is not.

We thus appear to be faced with the following vicious cycle.

a) We would like to pick an RGG for the strong component quickly, without searching all possible RGG's. To do so requires that we know what adorned subgoals can be captured.

b) To tell whether an adorned subgoal can be captured, we need to try all appropriate capture rules for that subgoal. The success of a capture rule test often depends on finding an appropriate RGG for the SCC, i.e., on picking the right order for subgoals in each occurrence of a rule.

The Forbidden-Adornments Assumption

Requirements (a) and (b) imply an exponential search through the space of possible strategies for an entire problem. To cut down on the exponent of the running time, we have adopted a heuristic that produces RGG's acceptable for a given capture rule, whenever they exist. When handling any one rule, with n subgoals, we generate at most $n(n-1)/2$ requests to evaluate whether some particular adorned subgoal can be captured, although in practice, the number of such requests is much closer to n, the minimum possible.†

In order that our strategy will produce an RGG whenever one exists, we need to make two assumptions:

1. If we can capture an adorned goal with a given capture rule, then we can also capture any "more bound" adorned goal with the same capture rule.
2. The set of RGG's for a given SCC that a particular capture rule can use is characterized by a set of *forbidden adorned goals*. Any RGG that has no adorned goals in the forbidden set can be used by the capture rule in question.‡

Example 7: The simplest example of this phenomenon is that the bottom-up capture rule for Datalog has no forbidden adorned goals, because it always works.

As a less trivial example, the capture rule described in Naughton [1987] for pushing selections through linear Datalog recursions requires that there be a binding on some argument of the recursive predicate. For example, in the transitive closure rules:

```
t(X,Y) :- e(X,Y).
t(X,Y) :- t(X,Z) & e(Z,Y).
```

we need a binding on some argument of t, so the set of forbidden goals is $\{t^{ff}\}$. □

Example 8: For a still more complicated example, consider the following, taken from Naish [1986]. An algorithm is given there that detects some cases when Prolog with function symbols converges (i.e., the top-down capture rule is applicable). The algorithm tries to find a set of arguments for each recursive predicate such that each time around the recursion, at least one argument has been given a subpart of its former value, and the remaining arguments have not changed.

Once the capture rule test, using Naish's or a similar algorithm, has determined such a set of arguments, it can attempt to construct an RGG, while forbidding any adorned subgoal in which these arguments are not bound. Consider the following definition of *merge*.

```
merge(X,[],X).
merge([],X,X).
merge([A|X],[B|Y],[A|Z]) :- A≤B & merge(X,[B|Y],Z).
merge([A|X],[B|Y],[B|Z]) :- A>B & merge([A|X],Y,Z).
```

The sets of arguments that work, as found by Naish's algorithm, are $\{1,2\}$ and $\{3\}$, as well as their supersets. That is, either the first and second arguments must be bound, or the third argument must be bound. Thus the forbidden adornments on *merge* are fff, fbf, and bff. □

To provide an initial guess at a good ordering for subgoals within a rule, we use the heuristic: select the subgoal with the most bound arguments; ties may be broken by selecting the subgoal with the fewest free arguments.§ This heuristic will suggest a candidate for the first adorned subgoal s^{α}. There are four possibilities.

---

† Since the number of subgoals in a rule tends to be small, say 2–5, it is not the difference between n or n^2 and $n!$ (the total number of orders) that makes the difference. Rather, there is an exponential speedup, as the small improvement for each rule accumulates over all rules in an RGG and all RGG's needed to solve a problem.

‡ The NAIL! system does provide a "hook" so that we can use capture rules for which this assumption does not hold, if needed.

§ We realize that this heuristic is not always optimal. However, the strategy-selection algorithm does not depend crucially on this or any other particular way in which subgoals are chosen for consideration.

1. Adorned goal s^α was previously installed in the rule/goal graph. Then we can assume it is capturable and proceed to pick the next goal in the ordering of subgoals for the present rule.
2. Predicate p is internal (belongs to the SCC), but s^α has not yet been installed in the RGG under construction. Temporarily install it, and then recursively construct the portion of the RGG accessible from s^α. If that construction is successful, make the installation of s^α in the RGG permanent. If the construction was not successful, then:
   a) Remove s^α from the RGG.
   b) Add s^α to the list of forbidden subgoals (for this RGG only, not for the capture rule itself).
   c) Select the next most desirable choice for the next subgoal in the rule ordering and repeat these five steps.
3. Predicate s is external (outside the SCC), but its capturability is not known. Recursively call the entire capture algorithm on s^α, determine whether it is capturable, and proceed as in (4), below.
4. Predicate s is external, and its capturability is known. If s^α is capturable, add s^α to the RGG, and proceed to select the next subgoal. Otherwise select the next most desirable choice for the next subgoal in the rule ordering.

Example 9: With *merge* rules as in Example 8, consider the adorned goal $merge^{ffb}$. Its capturability is unknown, so it is installed temporarily in the RGG, according to case (2). The first two *merge* rules can be added to the RGG immediately, since if the third argument in these rules is bound, all are.

In the third rule the heuristic chooses $A \leq B$ as the preferred subgoal because it and $merge(X,[B|Y],Z)$ both have one bound argument, but $A \leq B$ has fewer free arguments. The capturability of $\leq^{fb}$ is checked next. Since $\leq$ is outside the SCC of *merge* and is built-in, case (4) applies; presumably, only $\leq^{bb}$ is capturable.

Thus $\leq$ must be rejected as the first subgoal, and $merge^{ffb}$ tried instead. This adorned goal is in the SCC under construction, so case (1) applies: We assume it is capturable and that it binds its variables, B, X, and Y (Z was already bound). Now $A \leq B$ is reconsidered. Its adornment has improved to *bb*, which is capturable, so it is added to the RGG. The fourth rule is handled similarly. □

To summarize the NAIL! strategy-selection algorithm, we begin with an adorned goal p^α. We include that adorned goal in the RGG for the strongly-connected component of p, and proceed to add the rules for p^α to this RGG. We follow the points just outlined to select adornments for the subgoals of these rules, then consider the rules for these subgoals, and so on, until the RGG is complete or at some point we fail to find a possible ordering for the subgoals of some rule.

In the former case we have not only the RGG for p^α, but we have constructed RGG's, and by implication selected capture rules, for all of its subordinate goals. In the latter case, there is no RGG that will allow the particular capture rule in question to work, and we must consider other capture rules for p^α, if the system knows any. As other capture rules may have smaller sets of forbidden subgoals, we may hope that another capture rule will yet be successful.

VI. DETAILS OF THE STRATEGY SELECTION ALGORITHM

There are four routines, **capture**, **genRGG**, **processAR**, and **selectNSG** whose interaction forms the kernel of the strategy selection mechanism. These routines call upon a collection of capture rules, that is, pairs of test and substantiation algorithms; the test algorithms help make strategy decisions, and the substantiation algorithms generate ICODE when called upon to do so.

We assume that capture rules obey the forbidden adornments assumption, which enables them to use the procedure **genRGG** that implements the construction described above. However, we have made provision for capture rules whose legal RGG's are not characterized by forbidden adornments. These capture rules would have to construct an RGG their own way; they could not use

genRGG. Certain of the capture rules, such as those for built-in predicates or EDB predicates, likewise do not use **genRGG**, because we have no need of a RGG for these simple predicates. Figure 5 shows how the system goes about capturing an adorned goal. *Slanted* names are data; names in **boldface** are parts of the program.

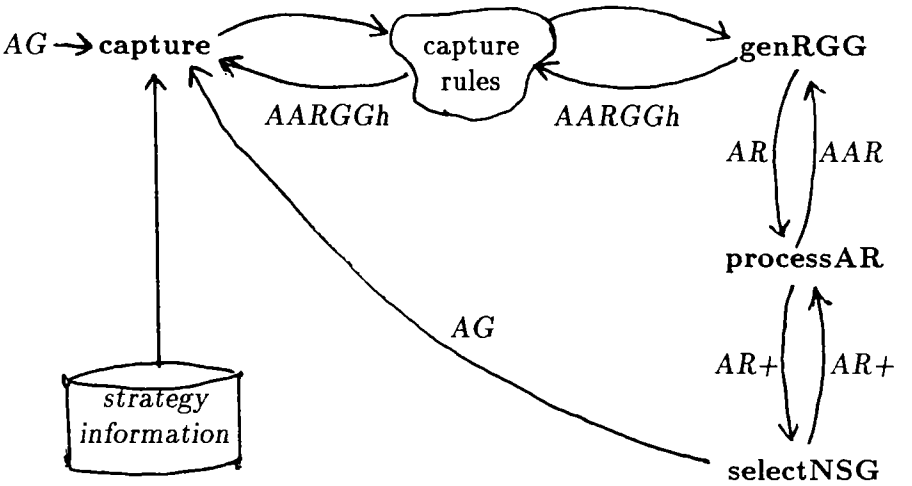


Fig. 5. Capturing an adorned goal.

The Procedure "capture"

First, **capture** decides which capture rule to use. We divide all SCC's into *classes*, depending on the form of their rules or other factors. The classes are easily modifiable, but the following are currently in use.

1. EDB predicates.
2. Built-in arithmetic predicates, e.g., $X < Y$.
3. Negated subgoal. Recall that negation can only be used in a stratified way, and after preprocessing rules, all negations appear in the context of a rule p :- ¬q, where q is in a "lower" SCC than p.
4. Nonrecursive datalog. We distinguish "datalog" rules by their absence of function symbols.
5. General, nonrecursive; i.e., function symbols are allowed.
6. Linear recursive datalog. "Linear" means that there is at most one subgoal of any rule that is mutually recursive with the head of the rule.
7. Nonlinear recursive datalog.
8. Recursive nondatalog.

Each category has an associated list of capture rules that can be applied to predicates in that class. The capture rules for each category are stored in order of ease of substantiation, easiest first, so we try each in turn.

The set of possible capture rules, like the classes, is extensible. For an idea of the kinds of capture rules that may be implemented eventually, Bancilhon and Ramakrishnan [1986] survey algorithms for datalog; Henschen and Naqvi [1984], Bancilhon et al. [1986], and Naughton [1987] are potential algorithms for the linear datalog case, Ullman [1985], Afrati et al. [1986], Naish [1986], and Ullman and Van Gelder [1985] give algorithms suitable for some nondatalog recursions, and Beeri and Ramakrishnan [1987] give algorithms that hold promise for nonlinear recursive datalog.

Once we have a capture rule to work on, we consult the stored strategy facts. If we already have discovered that we can capture the goal with the same adornment or a more general adornment, we can use the AARGGh stored. If we have found already that it is not possible to capture the

goal with the same or less general adornment, we do not need to invoke the capture rule; we try the next capture rule or report failure if there are no other rules).

The Procedure "genRGG"

The adorned goal is passed to the capture rule. Many capture rules need to construct annotated, adorned rule goal graphs, so a utility to form these graphs (**genRGG**), is provided. Its parameters are the adorned goal, the current SCC (strongly connected component of the predicate graph), and a list of forbidden adorned goals, created by the capture rule.

Procedure **genRGG** returns an annotated, adorned, rule-goal graph (AARGGh), which is represented by a list of arcs. There is no need to annotate the adorned goals in an AARGGh, since the AARGGh only covers one SCC. A strategy tells how to capture a whole SCC, so it will be the same for each goal in the SCC. The strategies for adorned goals outside of the SCC that are subgoals of the rules in it are contained in the annotated, adorned goals (AAG's) that are part of the annotated, adorned rules (AAR's); an AAR is a rule whose subgoals have been annotated and adorned, i.e., made into AAG's.

Then, **genRGG** takes the adorned goal, and constructs adorned rules for all the rules matching it and the other goals in its SCC. An adorned rule is formed by considering the head of the rule, the adorned goal, and the variables in the rule (remember that a rule adornment gives the bound/free status of the variables in the rule, not the arguments). These are passed, along with the list of forbidden adornments, to **processAR**. That procedure returns an AAR, giving both adornments and strategies for the various subgoals of the rule.

As some of the subgoals may be internal (within the current SCC), **genRGG** may have to consider new adorned goals and their rules, thus repeatedly invoking **processAR**. However, there are only a finite number of internal predicates and only a finite number of adornments that any one predicate can take, so the process must eventually construct an entire AARGGh or fail.

The Procedure "processAR"

Procedure **processAR** decides on an ordering for the subgoals of the adorned rule using the forbidden list, calling **selectNSG** to determine which subgoal to process next. As **processAR** works, it constructs a sequence of *augmented, adorned rules* (AR+), which are adorned rules with the subgoals partitioned into a list of *processed* (selected) subgoals and an unordered set of *unprocessed* (not yet selected) subgoals. Initially, all the subgoals are in the unprocessed set. All adorned goals on the processed list are either external and capturable, or internal. Further, no such subgoal can be the same as, or less bound than, any AG on the forbidden list.

Initially, the AR+ has all the subgoals in the unprocessed set. Repeatedly, **processAR** passes an AR+ to **selectNSG** and receives an updated AR+ back, which will have one more processed subgoal.

The Procedure "selectNSG"

The procedure **selectNSG** chooses the next subgoal, from among the unprocessed subgoals in the rule, using the SCC, the forbidden adornment list and **capture** to help make its decision. The decision is returned via an updated AR+, that includes the selected subgoal at the end of the list of processed goals. Procedure **selectNSG** uses a most-bound-first heuristic to consider candidates for selection. The subgoal returned may not be forbidden, and must either be internal or be external and capturable. If internal, we return an adorned goal, i.e., one with no annotation. If the subgoal is external, then we return an AAG, with the annotation giving the strategy whereby it was captured. Procedure **capture** is called recursively from **selectNSG** to find strategies for subgoals in lower SCC's.

VII. CURRENT STATUS

As of when this paper was written, October, 1987, the front end—rule preprocessor, strategy selector, and strategy database—have been implemented in Prolog. An interpreter for the ICODE, written in C, exists; it translates ICODE statements to SQL calls when needed, and executes them. We are in the process of putting the prototype together by writing the first capture rules, basic bottom-up and top-down methods, and integrating them with the system.

REFERENCES

Afrati, F., C. H. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv, and J. D. Ullman [1986]. "Convergence of sideways query evaluation," *Proc. Fifth ACM Symposium on Principles of Database Systems*, pp. 24–30.

Bancilhon, F., D. Maier, Y. Sagiv, and J. D. Ullman [1986]. "Magic sets and other strange ways to implement logic programs," *Proc. Fifth ACM Symposium on Principles of Database Systems*, pp. 1–15.

Bancilhon, F. and R. Ramakrishnan [1986]. "An amateur's introduction to recursive query-processing strategies," *ACM SIGMOD International Symposium on Management of Data*, pp. 16–52.

Beeri, C. and R. Ramakrishnan [1987]. "On the power of magic," *Proc. Sixth ACM Symposium on Principles of Database Systems*, pp. 269–283.

Henschen, L. J. and S. A. Naqvi [1984]. "On compiling queries in first-order databases," *J. ACM* **31**:1, pp. 47–85.

Minker, J. [1988]. *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, Los Altos. See the articles by Lifschitz, Przymusinski, Van Gelder, and by Apt, Blair, and Walker.

Morris, K., J. D. Ullman, and A. Van Gelder [1986]. "Design overview of the NAIL! system," *Proc. Third Intl. Conf. on Logic Programming*, pp. 554–568.

Naish, L. [1986]. "Negation and control in Prolog," Lecture Notes in Computer Science **238**, Springer-Verlag, New York.

Naughton, J. F. [1987]. "One-sided recursions," *Proc. Sixth ACM Symposium on Principles of Database Systems*, pp. 340–348.

Ullman, J. D. [1985]. "Implementation of logical query languages for databases," *ACM Trans. on Database Systems* **10**:3, pp. 289–321.

Ullman, J. D. [1986]. "NAIL! ICODE summary: C/SQL version," unpublished memorandum, Stanford Univ., Dept. of CS.

Ullman, J. D. [1988]. *Principles of Database and Knowledge-Base Systems*, Vol. I, Computer Science Press, Rockville, Md.

Ullman, J. D. and A. Van Gelder [1985]. "Testing applicability of top-down capture rules," STAN–CS–85–1046, Dept. of CS, Stanford Univ. To appear in *J. ACM*.

A Data/Knowledge Base Management Testbed

Raja Ramnarayan

Honeywell Corporate Systems Development Division
1000 Boone Avenue North
Golden Valley, Minnesota 55427

Tel: (612)-541-6844
ramnarayan%hi-csc@umn-cs.arpa

Hongjun Lu

Department of Information Systems and Computer Science
National University of Singapore
Singapore 0511

1. Introduction

Honeywell's Corporate Systems Development Division has been contracted by RADC under the Very Large Parallel Data Flow (VLPDF) program to investigate techniques for the design of data/knowledge base management systems (D/KBMSs). VLPDF is a two year program scheduled to be completed at the end of 1987.

We have adopted the logic programming paradigm for realizing a D/KBMS and henceforth, we will use the terms logic database system and D/KBMS interchangeably. The same is also true for the terms logic database and data/knowledge base (D/KB).

A data/knowledge base is comprised of two components: the extensional database (EDB) and the intensional database (IDB). The EDB consists of the base predicates and is also called the *database*, while the IDB consists of rules expressed in the form of Horn clauses and is also called the *rulebase*.

A common approach to D/KBMS design is to compile IDB queries into a program that computes the least fixed point (LFP) of a relational algebra expression and that accesses only the EDB [Banc86, Hens84, Ullm85]. EDB queries, on the other hand, are handled using traditional database query compilation and optimization techniques.

The D/KBMS architecture in the compilation approach consists of two layers:

- Knowledge Manager (KM), which compiles an IDB query, using appropriate optimizations, into a program that accesses the EDB, and
- Data Base Management System (DBMS), which executes the program generated by the Knowledge Manager.

The two layered architecture above is really a reference architecture. D/KBMSs with widely varying performance are realized via different choices for the KM optimization and LFP implementation strategies and the KM/DBMS interface.

This work was supported in part by Rome Air Development Center under the Very Large Parallel Data Flow program, contract number F30602-85-C-0215. The technical contract monitor for this program is Dr. Raymond Liuzzi.

We have designed and implemented a data/knowledge base management testbed on top of a commercial relational database system. The Knowledge Manager in our testbed compiles pure, function-free Horn clause queries into embedded SQL programs, which are executed by the DBMS.

Our objective was to build a tool that would serve as both a demonstration and performance measurement and evaluation platform. As a demonstration platform, the testbed illustrates the motivation and basic functionality of a D/KBMS, the components of a D/KBMS architecture, alternative implementations of these components and their relative tradeoffs, and the factors contributing to D/KB query compilation and execution time. As a performance measurement and evaluation platform, the testbed allows us to make quantitative performance measurements and to study system performance sensitivity and behavior with respect to several parameters.

We point out that this testbed is not intended to be a high performance logic database system. In particular, the choice of SQL as the KM/DBMS interface significantly degrades performance. However, for the purposes listed above, the testbed has proven to be a very valuable tool.

This paper describes the testbed architecture and is organized as follows. Section 2 describes the testbed architecture, section 3 D/KB query compilation, and section 4 status.

2. Testbed Architecture

The overall configuration of the testbed is shown in figure 1. The testbed consists of four components: User Interface, Knowledge Manager, DBMS, and Run Time Library.

2.1. User Interface

The main options provided by the User Interface are:

- Enter rules, which allows the user to enter a set of Horn clauses into the workspace D/KB (see section 2.2 below for the definition of workspace and stored D/KBs).
- Enter query, which allows the user to enter and compile a Horn clause query.
- Execute query, which allows the user to execute a previously compiled query.
- Update stored D/KB, which allows the user to update the stored D/KB with rules and facts from the workspace D/KB.

2.2. Knowledge Manager

The Knowledge Manager accepts Horn clauses and queries from the User Interface and compiles queries into code fragments. It consists of the following components: Rule Parser, Workspace D/KB Manager, Stored D/KB Manager, Semantic Checker, Optimizer, and Code Generator. The KM architecture is shown in figure 2. The circles in this figure represent data structures and the boxes, components.

The distinction between the workspace and stored D/KBs is best illustrated via a typical session. The user enters a set of rules and facts into the *workspace D/KB*, which is a private, memory resident environment. The rules in the workspace may refer to rules and facts in a shared disk resident repository, called the *stored D/KB*. The user then issues a query and if he is satisfied that the rules and facts he entered into the workspace are correct, he issues an update request, which updates the stored D/KB with rules and facts from the workspace D/KB.

In this section, we describe the KM components. The control flow among them during D/KB query compilation is described in section 3.

2.2.1. Rule Parser

The Rule Parser generates an internal representation of Horn clauses. This representation consists of two hash tables, rules and predicates, into which information contained in the source

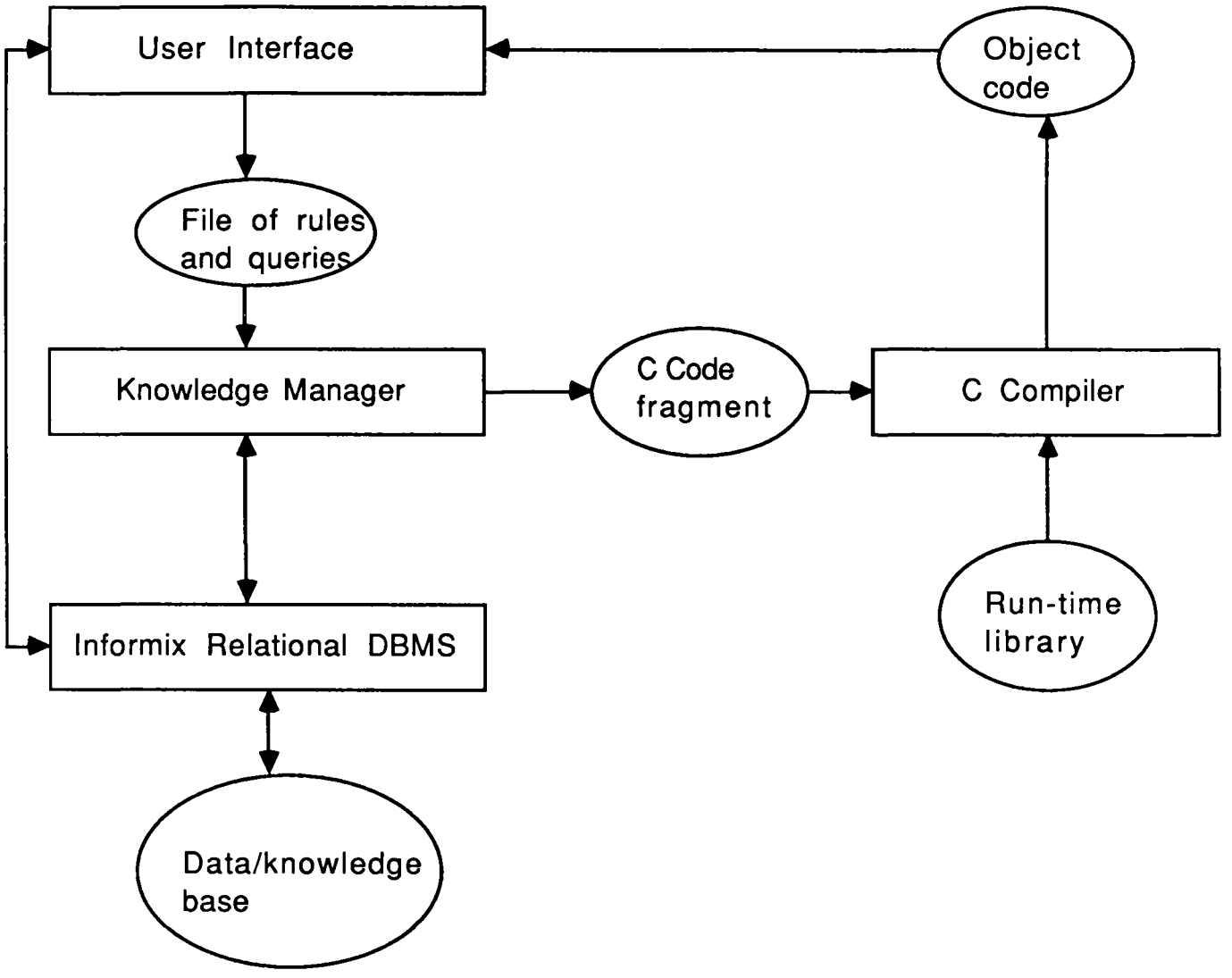


Figure 1: Testbed Architecture

form of the Horn clauses is loaded.

2.2.2. Workspace D/KB Manager

In the workspace, rules are represented as a predicate connection graph (PCG) [McKa81]. A PCG is a directed graph that represents the relationships between head predicates and body predicates. Nodes in the PCG represent predicates, while edges arise from rules. If there is a rule of the form, $p \leftarrow p_1, p_2, ..., p_n$, there is a directed edge from p to each of the p_i's.

We give some additional definitions below. A derived predicate q is *reachable* from a derived predicate p if

(i) q is in the body of a rule having p as its head, or

(ii) q is in the body of a rule having s as its head and s is reachable from p.

Two derived predicates p and q are *mutually recursive* if they are reachable from each other. A predicate p is *recursive* if it is reachable from itself. A rule $p \leftarrow p_1, p_2, ..., p_n$ is called a *recursive rule* is there exists a p_i in the body that is mutually recursive to p. Two rules are mutually recursive if the predicates in their heads are mutually recursive. It can be easily shown that

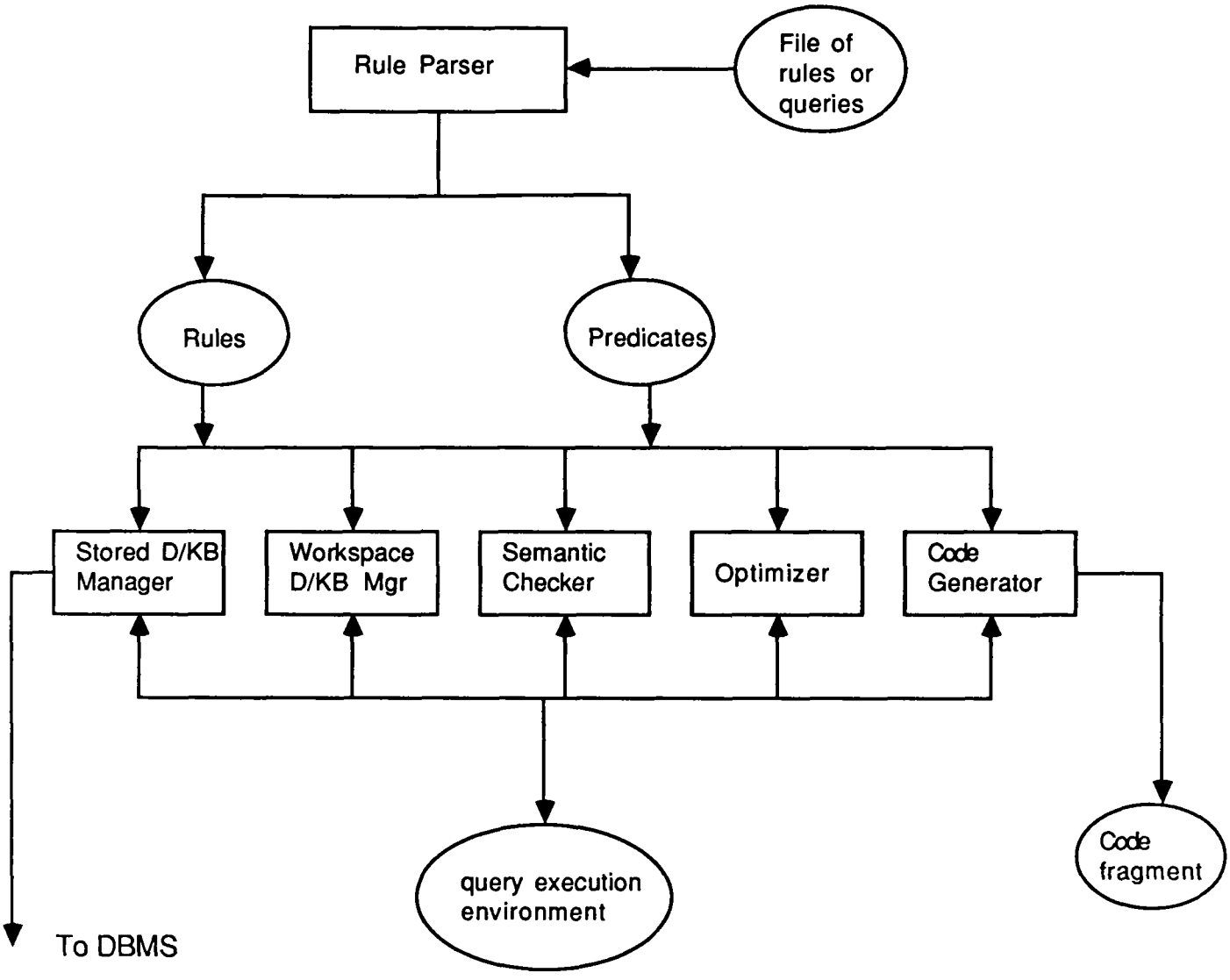


Figure 2: Knowledge Manager Architecture

mutual recursion is an equivalence relation on the set of derived predicates and the set of rules. Mutual recursion partitions the set of derived into disjoint blocks of mutually recursive predicates. The predicates in a block must be evaluated as a whole. Mutual recursion groups together rules needed to evaluate the predicates in a block.

The blocks of mutually recursive predicates are the *strongly connected components* of the PCG. A strongly connected component, also called a *clique*, of a graph is a set of nodes such that there is a directed path between each pair of nodes.

In the context of D/KB query processing, we will use a somewhat broader definition of a clique. Here, by clique we will mean a set of mutually recursive predicates as well as the rules needed to evaluate these predicates. Obviously, some of these rules will be recursive. The rest are called *exit rules*. Exit rules ensure that the evaluation of a recursive predicate terminates. Therefore, every clique must have at least one exit rule.

Clique evaluation, i.e., evaluation of mutually recursive predicates involves computing the LFP of a system of recursive equations. The algorithms for LFP computation are included in the Run Time Library of the testbed (see section 2.3). On the other hand, evaluating non-recursive predicates is accomplished via a straightforward compilation to relational algebra.

The cliques need to be evaluated in a certain order to ensure that a rule is evaluated only after the predicates in its body are solved. To determine this order, the Workspace D/KB Manager constructs the *evaluation graph*. This is a directed graph whose nodes are either derived predicates or cliques. There are four types of directed edges: (1) $P \rightarrow C$ indicates that some predicate in the clique C appears in the body of a rule defining P, (2) $C \rightarrow P$ indicates that P appears in the body of a rule defining some predicate of C, (3) $P_1 \rightarrow P_2$ indicates that P_2 appears in the body of a rule defining P_1, and (4) $C_1 \rightarrow C_2$ indicates that some predicate of C_2 appears in the body of a rule defining some predicate of C_1. The evaluation graph is essentially the PCG with the base predicates removed and the predicates of a clique collapsed into a single node. Thus, the predicate nodes of the evaluation graph are the nonrecursive predicates of the PCG. Also, while the PCG may be cyclic, the evaluation graph is acyclic. The Workspace D/KB Manager determines the order in which to evaluate the cliques by performing a topological sort of the evaluation graph. The resulting total order is called the *evaluation order list*.

With this background, we can now state the basic functions provided by the Workspace D/KB Manager: (1) determine all predicates reachable from a given predicate, (2) find the cliques in the workspace D/KB rules, and (3) generate the evaluation order list.

2.2.3. Stored D/KB Manager

The Stored D/KB Manager provides the following functions:

- Extract rules from the stored D/KB needed to solve a given set of predicates.
- Read the IDB and EDB data dictionaries during type checking, which is the process where the types of the columns of the derived predicates are inferred (see the Semantic Checker, section 2.2.4 below).
- Update the stored D/KB with rules and facts from the workspace D/KB.

The rule storage structures in the stored D/KB consists of four relations: *isystables, isyscolumns, irulesource,* and *ireachablepreds*. The first two relations, *isystables* and *isyscolumns*, are the IDB data dictionary. They contain the types of the columns of the derived predicates.

The rule storage structure tables have the following schema:

isystables(*tablename char*, *tableid integer*)

isyscolumns(*tableid integer*, *colname char*, *colnumber integer*, *coltype integer*)

irulesource stores for each derived predicate p, the rules defining p. It has the following schema:

irulesource(*headpredname char*, *rule char*)

ireachablepreds is the transitive closure of the PCG of the rules stored in *irulesource* and constitutes the compiled form of these rules. It stores for each derived predicate p all the predicates reachable from p. It has the following schema:

ireachablepreds(*frompredname char*, *topredname char*).

ireachablepreds allows very efficient retrieval of the relevant rules from the stored D/KB. For example, suppose the stored D/KB contains the following rules:

R_1: $p(X, Y) \leftarrow a(X, Z), q(Z, Y)$.

R_2: $a(X, Y) \leftarrow b_1(X, Z), a(Z, Y)$.

R_3: $a(X, Y) \leftarrow b_2(X, Y)$.

R_4: $q(X, Y) \leftarrow c(X, Y)$.

R_5: $c(X, Y) \leftarrow b_3(X, Y)$.

R_6: $m(X, Y) \leftarrow b_4(X, Y)$.

where the b_i's are base predicates. Then retrieving all the rules needed to solve the query

$query(X, Y) \leftarrow p(X, Z), m(Z, Y)$.

is accomplished via the following SQL query:

```
SELECT irulesource.rule
FROM irulesource, ireachablepreds
WHERE (ireachablepreds.toppredname = irulesource.headpredname OR
       ireachablepreds.frompredname = irulesource.headpredname) AND
       ireachablepreds.frompredname = "p" OR
       ireachablepreds.frompredname = "m"))
```

This query retrieves rules R_1 through R_6 above. To speed up the execution of this query, we place indexes on *irulesource* and *ireachablepreds*.

During updates, the Stored D/KB Manager only updates the rule storage structures; it doesn't check the workspace D/KB rules and facts against integrity constraints that may be associated with the stored D/KB.

2.2.4. Semantic Checker

The Semantic Checker performs two kinds of checks. The first is to check for each derived predicate reachable from the query, whether there is a rule defining it and the second is the type check for inferring the types of the columns of the derived predicates. We have developed a type inference algorithm, whose flavor we give below.

The type of each column of a base predicate is fixed at the time it is created. The type of the columns of the derived predicates is inferred from the rules. For example, in the rule $p(X, Y) \leftarrow b(X, Y)$, the type of the first (respectively, second) column of p is the same as that of the first (respectively, second) column of b. Type checking involves inferring the types of the derived predicates and also checking whether the same types are inferred from all the rules defining p. This is easy to do for nonrecursive predicates. However, for recursive predicates, we need to loop till either closure is reached or there is a type mismatch.

2.2.5. Optimizer

D/KB query processing performance strongly depends upon the efficiency of LFP computation. The Optimizer rewrites the rules relevant to the query into a new set of rules that is equivalent to the original set but whose LFP computation is more efficient. The main idea in D/KB query optimization is the use of sideways information passing (sip) to restrict the computation to tuples that are related to the query. Our testbed uses the generalized magic sets strategy proposed by Beeri and Ramakrishnan [Beer87]. Our original work here is an algorithm for generating a sip strategy. We will describe this algorithm in a more comprehensive paper that we are in the process of preparing.

2.2.6. Code Generator

The Code Generator generates a code fragment, which basically loads certain data structures in the object program with query specific information. These data structures contain information similar to the nodes of the evaluation order graph. For predicate nodes, the code fragment loads the predicate name, schema information (name and type of each column), and the

SQL query to evaluate the body of each rule in which the predicate appears as head. For clique nodes, the code fragment loads the same information, except that it differentiates between exit rules and recursive rules.

2.3. Run Time Library

In general, evaluating the mutually recursive predicates in a clique will involve finding the LFP of a set of recursive equations of the form:

$$r_1 = f_1(r_1, \ldots, r_n)$$

$$\vdots$$

$$r_n = f_n(r_1, \ldots, r_n)$$

The LFP is guaranteed to exist since the functions f_i are all monotone in the case of Horn clauses.

The Run Time Library contains our implementation of two bottom-up strategies for computing the LFP of the above system of equations: naive evalaution [Banc86] and semi-naive evaluation [Banc85]. As we mentioned in the introduction, SQL is not the KM/DBMS interface to use when designing a high performance D/KBMS, since relational algebra cannot express LFP queries [Aho79]. Consequently, the above system of equations is evaluated as an application program and there is not much scope for optimization here; several temporary tables are created and dropped during each iteration, which introduces a lot of overhead.

The code fragment produced by the Knowledge Manager is compiled and linked with the run-time library to produce the object code, which is executed by the User Interface as an application program against the DBMS to give the query results.

3. D/KB Query Compilation

In this section, we describe the control flow among the KM components during query compilation.

The Workspace D/KB Manager determines all predicates reachable from the query. In general, there may be rules defining these predicates in the stored D/KB. The Stored D/KB Manager extracts these rules using the SQL query described in section 2.2.3 and loads them into the workspace.

At this point, all the rules needed to solve the query (from both the workspace and stored D/KBs) are present in the workspace. The Workspace D/KB manager, once again, determines all the predicates reachable from the query. This will now include workspace as well as stored D/KB predicates.

The Optimizer rewrites the rules relevant to the query as per the generalized magic sets strategy. It generates three sets of rules in the workspace: the adorned, magic and modified rules. It also generates an adorned version of the query.

The Workspace D/KB Manager then finds the cliques in the adorned rules and uses this information to generate the evaluation order list.

The Semantic Checker now performs the two tests described in section 2.2.4. Finally, the Code Generator generates the code fragment for each entry (clique or non-recursive predicate) in the evaluation order list.

4. Status

We have implemented a version of the testbed, which includes all the KM components described in this paper except the Optimizer. This testbed is implemented using the Informix relational DBMS.

We have run several performance measurement experiments on the testbed. These experiments determine the effect of various D/KBMS parameters such as IDB and EDB size and EDB type (list, tree, cyclic graph, acyclic graph, etc.) on query compilation and execution times. They also compare the relative performance of naive vs semi-naive LFP evaluation, source vs compiled form storage of rules in the stored D/KB, and the performance improvement made possible using the generalized magic sets strategy. This last is accomplished by hand coding the adorned, magic, and modified rules. A comprehensive paper detailing the testbed architecture, performance measurement and evaluation, analysis, and conclusions for D/KBMS design is forthcoming.

5. References

References

Aho79. A. Aho and J. Ullman, "Universality of Data Retrieval Languages," *Proceedings of 6th POPL*, 1979.

Banc85. F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems Integrating Database and AI Systems*, ed. Brodie and Mylopoulos, Springer-Verlag, 1985.

Banc86. F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM-SIGMOD*, May 1986.

Beer87. C. Beeri and R. Ramakrishnan, "On the Power of Magic," *Proc. of 6th ACM SIGMOD-SIGACT symposium on Principles of Database Systems*, 1987.

Hens84. L. J. Henschen and S. Naqvi, "On Compiling Recursive Queries in First Order Databases," *JACM*, vol. 31, no. 1, pp. 47-85, January 1984.

McKa81. D.P. McKay and S.C. Shapiro, "Using Active Connection Graphs for Reasoning with Recursive Rules," *Proc. 7th International Conference on Artificial Intelligence*, pp. 368-374, August 1981.

Ullm85. J.D. Ullman, "Implementation of Logical Query Languages for Databases," *ACM TODS*, vol. 10, no. 3, pp. 289-321, September 1985.

An Overview of the LDL System

Danette Chimenti, Tony O'Hare, Ravi Krishnamurthy,
Shamim Naqvi, Shalom Tsur, Carolyn West and Carlo Zaniolo

MCC, Austin, Texas

1. Introduction

The Logic Data Language LDL is a declarative language for data–intensive and knowledge–based applications built upon the amalgamation of Logic Programming and Relational Databases. Thus, the LDL system supports rule based programming, pattern matching and inferencing as in Prolog, along with transactions, recovery, integrity and secondary storage management as in a DBMS. Moreover, while Prolog users have to order rules and clauses carefully to ensure termination and efficient execution, LDL users are relieved of this responsibility that is taken over by the system — as in relational DBMSs, it is done by the query optimizer. Therefore, the LDL system offers all the benefits of a database language, including the elimination of the "impedance mismatch" between the programming language and the database query language currently besetting the development of data intensive applications.

The design and development of the LDL system has posed several research and design challenges, which can be roughly divided into five problem areas. The first area is *Language Design*, where the main problems follow from the need for defining simple and expressive constructs to handle *Negation*, *Sets* and *Updates* while preserving the syntactic structure and the model theoretical and fixpoint semantics of Horn Clauses. [VK 77, AE 82] These problems are discussed in section 2.

The second problem area pertains to the choice of the execution model and abstract machine most suitable for the intended application domain and system architecture. As discussed in Section 3, our execution model and abstract machine target language are based on relational algebra. The third problem area pertains to the compilation of LDL into this target language. The rule rewriting techniques used for this purpose are described in Section 4, while Section 5 describes the novel techniques used to support set terms.

The fourth area of research pertains to the difficult problem of devising, at compile time, a safe and efficient execution plan for a LDL query. Section 6 —Optimizer Design– discusses these issues. The final set of problems includes the integration of these techniques and the design and development of a robust system; these are briefly described in Section 7.

2. Language Design

The language design of LDL [TZ86, BNST87] reflects a compromise between two conflicting trends:

1. The desire to create a purely declarative language which provides full support for Horn clause logic as well as various extensions to this logic.

2. A practical need for the support of certain features, updates in particular, which are intrinsically procedural in nature.

Support for Horn clause logic implies that the query response must be consistent and complete i.e., for a given LDL program and a given query, the set of all tuples of bindings which can be logically concluded from the underlying Herbrand base of the program are returned [LLOYD84]. The language is thus oriented towards

applications which expect set-at-a-time results as opposed to other programming languages e.g., PROLOG, which returns a tuple at a time.

LDL contains, in addition to Horn clause logic, a number of features which have been included to provide an enriched functionality to the user. These comprise a support for negation in the logical sense, an enriched knowledge representation capability over a universe which contains set objects, and procedural attachments to rules, primarily (but not exclusively) for the purpose of updates. The inclusion of these features required the formulation of a semantics which cannot be captured anymore in the familiar notions of an Herbrand universe. The reader is referred to [BNRST87] for details. We will now briefly elaborate on each of these features and demonstrate them by means of examples.

The following example derives an *exclusive_pairs* relation from an *edge* base relation which represents a directed graph. The tuple *(a,b)* is included in *edge* if the graph contains a directed arc from node *a* to node *b*.

Example: Use of Negation.

reachable(X, Y) ← *edge(X,Y).*
reachable(X, Y) ← *edge(X, Z), reachable(Z, Y).*

exclusive_pairs(X, Y, Z) ← *reachable(X, Y), ~ reachable(Z, Y).*

The *reachable* relation derives the transitive closure over the *edge* relation. It contains thus all pairs of nodes such that a path exists between them in the graph. The *exclusive_pairs* relation contains all pairs of nodes such that a path exists between them *except* those pairs which can be reached from node Z (the binding to this argument is assumed to be given). The logical interpretation is thus a *difference* between the set of all pairs and the set of pairs associated with the given Z-node.

The mechanism employed by the compiler to enable an interpretation of this type is that of *stratification* [NAQ86, ABW86, VG86]: all of the negated predicates in a rule body must be first derived in their positive form i.e., they must appear in the head of another rule, prior to negation. This condition, which imposes a partial order of evaluation over the program rules, is enforced by the compiler. Programs which comply with this restriction are called *admissible programs*. Not all programs are admissible as the following example will demonstrate.

Example: Inadmissible Program.

int(0).
int(succ(X)) ← *int(X).*

even(0).
even(succ(X)) ← *int(X), ~ even(X).*

In this program *even* is defined in terms of itself. The program is thus unstratifiable.

As we have mentioned, the underlying LDL universe has been enriched to include set objects. Consequently, the language contains *set terms* which are used to match set objects under commutativity e.g., *{a,b}* = *{b,a}* and idempotence e.g., *{a,a}* = *{a}*. However associativity e.g., *{a,{b,c}}* = *{a,b,c}* is not included in these mechanisms and sets with different levels of nesting are recoginzed as different objects. We will elaborate on this subject when we discuss the compilation of rules containing set terms. The following example demonstrates the use of set terms.

Example: Set terms.

h(X) ← *friends({X,Y,john}), X ≠ john, nice(X).*

In this example the base relation *friends* is over sets. The set term will match sets of cardinality greater than or equal to 1 and less than or equal to 3 containing the element *john*. Thus, the set object *{jack, john}* will

be matched by the set term and result in bindings *{X/jack, Y/jack}* provided that *nice(jack)* is in the database.

Another form of set use is by *grouping* — the derivation of an unnormalized relation in which all values which can be grouped by some key have been collected in a set object. The following example derives the unnormalized *suppliers_set* relation from the normalized *part_supplier* base relation.

Example: Set grouping.

suppliers_set(Part, <Supplier>) ← *part_supplier(Part, Supplier).*

The < > brackets in the head denote a set. For each binding to *Part* all of the bindings for *Supplier* supplying that part are grouped together in a set.

While LDL is a declarative language, it is occasionally necessary for the user to impose his own control over the order of evaluation instead of leaving it to the system. This is of particular importance when updates are considered. The semantics of queries in the context of a purely declarative language is insensitive to the particular order chosen by the system to evaluate the query. For updates however this is not true in general. Consider the following example in which the base relation is of the type *emp(Name, Dept, Salary).*

Example: A rule containing updates.

happy_emp(X, NewSal) ← *emp(X, database, Sal), NewSal = Sal * 1.1,*
(-emp(X, database, Sal), +emp(X, database, NewSal)).

The derived relation contains happy employees — those who work in the database department and who received a 10% salary raise (the + and - symbols denote insertion and deletion).The body of this rule consists of two phases: a *query phase* which "marks" those tuples that will be updated and a *procedure phase* which performs the update. The order in which these phases occur is not arbitrary — the query phase must occur before the procedure phase. Furthermore, a *well-formedness* condition [Nak87] which must be observed ensures that all of the variables occurring in the procedure phase are covered by the variables appearing in the query phase and are thus bound. Within the procedure phase, further analysis is required to determine whether the procedure satisfies the *Church-Rosser* property i.e., the procedure produces the same answer for every permutation of its components. In the case this property holds then the compiler may choose any order of execution — the one which is most efficient. When the property does not hold then the procedure is executed in the order specified by the programmer i.e., left to right. For the example above, the Church-Rosser property does not hold, as can be seen when *emp* contains two tuples of the form *emp(smith, database, 10K)* and *emp(smith, database, 11K).* Delete before insert will result in *emp(smith, database, 11K), emp(smith, database, 12.1K)* while on the other hand, insert before delete will result in *emp(smith, database, 12.1K)* since smith's new salary equals the old salary of the other smith thus deleting *emp(smith, database, 11K)* as well as *emp(smith, database, 10K)* . This problem bears resemblance to the serializability requirement in transaction management systems.

3. Execution Model and Target Language

Current implementations of Prolog are based on SLD resolution [LLOYD84], with leftmost goal expansion, which was further refined into the well-tuned execution model of the Warren Abstract Machine [War85]. This approach is not best suited for data intensive applications where operations on large sets of data and access to secondary storage become the main concern. Thus a target language based on relational algebra was used instead. For instance, the following Horn Clauses defining a grandmother

? grandma(X,Y).
grandma(X, Z) ← *mother(Y, Z), parent(X, Y).*
parent (X, Y) ← *mother(X, Y).*
parent (X, Y) ← *father (X, Y).*

can be mapped into the expression below (F and M denote the relations containing the base facts for father and mother while GM is the grandma derived relation):

$$GM = (M \bowtie (F \cup M)).$$

A straightforward computation of this relational algebra expression can be performed in a *bottom up* (from the database to the query) execution order. However, bottom up is only one of the many execution strategies used in the LDL implementation. This flexibility follows via a *materialized* or a *pipelined* option for joins and unions. Materialized joins compute their whole operands before performing the operation. Pipelined joins apply the value obtained from a tuple in the left operand to select the matching tuples in the right operand. If this operand is a pipelined union then the selection distributes over the operands of the union. This approach allows for a wide variety of execution styles including the sideways information passing discussed in [MNSUG] and Prolog forward chaining search strategy. Various extensions to relational algebra are needed to accommodate the additional expressive power of LDL over relational calculus. The simple extensions described in [Zan85] support the retrieval and manipulation of complex terms. Recursion is supported via a fixpoint operator. The actual target language, called FAD [BBKU87], supports the basic operations just described, along with updates and other constructs found desirable in various situations. These include a construct used for the parallel execution of set aggregate operations and also procedures, conditionals and iterators— the latter used for implementing fixpoint operators.

4. Compilation

A first task of the LDL compiler is to parse the rule base and generate a Predicate Connection Graph based representation for these rules [KOT86]. The compilation proper begins with a query form — i.e., a query where mode declarations denote which arguments will be given and which will be derived at actual query time. Thus a relevant PCG denoting the set of rules needed to support this query form is constructed and the constant migration step for non–recursive predicates is performed, as described next. For instance, say that the query form

? grandma($X, Y).

is given, with *$X* denoting a *deferred constant* , i.e., a value to be specified at execution time. Then the given set of rules can be specialized as follows (further specializations, such as dropping *$X* from parent predicate, are here disregarded for the sake of simplicity):

? grandma($X,Y).
grandma($X, Z) ← *mother(Y, Z), parent($X, Y).*
parent ($X, Y) ← *mother($X, Y).*
parent ($X, Y) ← *father ($X, Y).*

Observe that this step may result in duplication of rules, as two predicates with different bound arguments (i.e., different bound–free adornments) must be supported independently. For instance, if mother were a derived predicate rather than a base one, then we would need a distinct set of rules for *mother($X, Y)* and for *mother(Y, Z)*. Straightforward *rule transformation techniques* are used to implement these transformations.

The optimizer takes the relevant PCG so transformed and rearranges and annotates it as to guarantee a safe and efficient execution (for queries matching the given query form). For the query at hand, for instance, the optimizer may specify an execution that amounts to the goal *parent($X, Y)* being materialized first, and this being then joined in a pipelined fashion with *mother*. Thus, the important choices made by the optimizer include that of the join methods (e.g., pipelining or materializing methods) and of fixpoint based methods to support recursion which is discussed next.

The fixpoint computation, improved through differential techniques yielding the so called *semi-naive fix-*

point algorithm [Banc85, BaRa86, ZaSa87], supports efficiently a query form where no argument is bound, such as the following one:

? ancestor(X,Y)
ancestor(X,Y) <– parent(X, Z), ancestor(Z, Y).
ancestor(X, Y) <– parent(X, Y).

However, for a query form such as

? ancestor($C, Y).

we would like to push the deferred constant *$C* into the recursive rule, in order to avoid constructing the whole relation when only ancestors of *$C* are needed. (A similar situation occurs whenever one wants to pipeline the *X*-values into an ancestor goal in a rule.) Unfortunately, the simple approach of substituting the deferred constant for the corresponding variables does not work for recursive predicates —in particular the case at hand. More complex approaches are needed and many have been proposed [AhUl79, BeRa87, BMSU86, BR86, GaDe86, HeNa84, KiLo86, RLK86, SaZa86a, SaZa86b, SaZa87a, Vie86]. LDL's implementation uses the following methods: *single fixpoint, counting* and *magic set.*

Pushing selection into recursive predicates by replacing variables with constants works in many situations of practical interest when used in combination with techniques for converting recursive rules into equivalent ones. For instance, for the previous query the recursive rule can be transformed into its right linear equivalent and then specialized by substituting *$C*:

? ancestor($C, Y).
ancestor($C,Y) ← ancestor($C, Z), parent(Z, Y).
ancestor($C, Y) ← parent($C, Y).

These rules, perfectly customized for the given query form, translate into a fixpoint computation that is safe and efficient. While the existence of equivalent rules where constant pushing works remains an undecidable problem [BKBR87], the LDL compiler is capable of recognizing and handling many such situations. The remaining situations are handled by either the counting or the magic set method, both of which implement recursion via a *pair* of fixpoint computations. The counting method is used for simple linear queries where there is no problem with cycles in the database [BR86], as there is ample evidence that it is more efficient than the magic set method for these simple situations[BR86,MPS87]. For complex recursive rules where the counting method becomes more complex and when dealing with cycles, the magic set method is used instead. The possibility of replacing both methods by an integrated method known as magic counting [SaZa86b] is also being investigated. Rule rewriting techniques are used to implement the methods here described and also the semi-naive fixpoint improvement.

5. Compilation of Rules Containing Set Terms

The compilation of rules containing set terms is consistent with the overall methodology adopted in LDL. The existing methods towards the matching and unification of sets [LC87] are all employed at run time and execute in times exponential in the size of the participating sets. The approach employed in LDL [STZ87] is of rewriting the program *at compile time* into an equivalent program in which each rule containing set terms has been replaced by a set of rules, each containing ordinary (i.e., non-set) terms only. The execution cost of a query, using the transformed program, is substantially lower than the cost of using the original program since now ordinary matching techniques can be used instead of set matching techniques. The additional cost incurred by the transformation of the program at compile time is thus amortized over many executions of a query. An additional advantage of this method is that the compile time analysis enables the employment of a number of optimization methods which cull in advance alternatives which can be proven to be unsuccessful as opposed to the run time methods which would explore all dead alleys. Let us return to our previous example with set terms,

h(X) ← friends({X, Y, john}), X ≠ john, nice(X) .

Using compile time methods this rule is rewritten as:

h(X) ← funnel_up_friends(set_of(X, Y, john)), X ≠ john, nice(X).

The *friends* relation in the original rule has been replaced by *funnel_up_friends*; The set term *{X, Y, john}* has been replaced by an ordinary complex term *set_of(X, Y, john)* in which *set_of* is a functor name, designated to sets. The set properties viz. commutativity and idempotence must now be "built in" the structure of the funnel-up rule which, in this case after compilation, looks as follows:

funnel_up_friends(set_of(Y, X, john)),

funnel_up_friends(set_of(X, Y, john)) ← friends(set_of(john, Y, X));
friends(set_of(Y, john, X));
friends(set_of(Y, X, john)).

funnel_up_friends(set_of(X, X, john)) ← friends(set_of(john, X));
friends(set_of(X, john)).

All of the potential matches for sets of cardinalities 2 and 3 have been included. Note that the singleton set case *i.e., funnel_up_friends(set_of(john, john, john)) ← friends(set_of(john))* has been excluded at compile time since the body restriction $X \neq john$ in the original rule dictates that the applicable sets have at least two members. The rule is represented in a compact format, called *Multi-Head-Multi-Body* in which $H_1, H_2, ..., H_n \leftarrow B_1; ...; B_m$ represents $m \times n$ rules of the format $H_i \leftarrow B_j$ $i=1, ...,n; j=1, ...,m$. Although this result may seem bulky, its execution is more efficient than set matching at run time. A crucial assumption, which makes these compile time methods possible, is that of a *standardized storage* of set objects. We assume that the sets in the *friends* relation are stored in an ascending ASCII order and instances of the type *friends(set_of(john, jack, joe))* do not exist since they violate this order (the order in this case would be *friends(set_of(jack, joe,john))*. Making use of this assumption enables the compiler to eliminate certain matching patterns in a multi head multi body rule which are guaranteed to fail at run time.

Assume now that the database contains the following facts:

friends(set_of(jack, jim, john)).

nice(jim).

nice(jack).

Then, a query of the form *?h(X)* would return *{X/jim, X/ jack}*. The first result is derived from α = *{X/jack, Y/jim}*. The second result is derived from β = *{X/ jim, Y/ jack}* and the fact that *set_of(jim, jack, john)* $=_{ci}$ *set_of(jack, jim, john)*. The notation $A =_{ci} B$ means that two terms are equal modulo commutativity and idempotence. Note that both α and β are funneled up to the heads of the first funnel up rule. As such, these heads represent the potential for commutativity in the bindings of *X* and *Y*.

6. Optimizer.

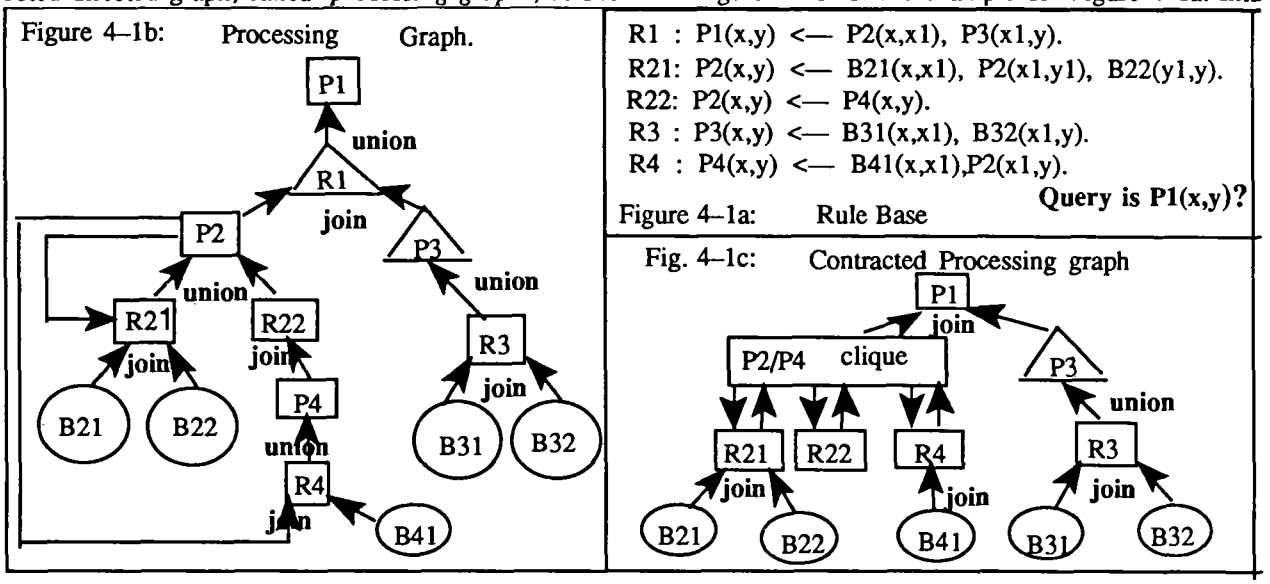
We define the optimization problem as the minimization of the cost over a given execution space (i.e., the set of all allowed executions for a given query). Any solution to this optimization problem can then be described along four main coordinates, as follows:
i) the model of an execution,PG, representing the relevant aspects of processing;
ii) the definition of the execution space, E, consisting of all allowable executions;

iii) the cost functions which associate a cost estimate with each point of E; and
iv) the search strategy to determine the minimum cost execution in the given space.

Obviously, the main trade-off in the optimizer design is that a very small execution space will eliminate many efficient executions, whereas a very large execution space will render the problem of optimization intractable, for a given search algorithm. We outline here the design of the execution model and thus the execution space, cost model and search algorithm.

6.1 Execution Model and execution space.

As LDL's target language is a "relational" algebra, an execution over this target language is modelled as a rooted directed graph, called *'processing graph'*, as shown in Figure 4–1b for the example of Figure 4–1a. Intu-



itively, nonleaf nodes (i.e., the nodes with non-zero in-degree) of this graph correspond to operators and the results of their predecessors are the input operands. (This is similar to the predicate connection graph[KT81] or the rule graph [Ull 85], except that we give specific semantics to the internal nodes and use a notion of contraction.) We map each AND node into a *join* and each OR node into a *union*. Recursion is implied by an edge to an ancestor or a node in the sibling subtree. A *contraction* of a clique is the extrapolation of the traditional notion of an edge contraction in a graph generating a strongly connected component of the graph which we call *recursive clique*. An edge is said to be *contracted* if it is deleted and its ends (i.e., nodes) are identified (i.e., merged). A clique is said to be *contracted* if all the edges of the clique are contracted. Intuitively, the contraction of a clique consists of replacing the set of nodes in the clique by a single node and associating all the edges in/out of any node in the clique with this new node, (as in Figure 4–1c), generically called *Contracted Clique node* (or *CC node)*. Intuitively, a CC node correspond to the fixpoint operation for the clique, whose operands are the results of the predecessors.

Associated with each node is a relation that is computed from the relations of its predecessors by doing the operation (e.g., join, union) specified in the label. We use a square node to denote materialization of relations and a triangle node to denote the pipelining of the tuples. Each interior node in the graph is also labeled by the method used (e.g., join method, recursion method etc.). The set of labels for these nodes is restricted *only* by the availability of the techniques in the system. The label for a CC node is to specify the choices for the fixpoint operation, which are the choices for Sideways Information Passing (SIPs) and recursive method to be used. The execution corresponding to a processing tree proceeds left to right, bottom–up execution.

Note that many logically equivalent (i.e., produce same result) processing trees, with very different costs, can be generated since each embodies critical decisions regarding the methods to be used for the operations, their ordering, and the intermediate relations to be materialized. The set of logically equivalent processing trees thus

defines the execution space over which the optimization is performed (using a cost model which associates a cost to each execution). The equivalence of two processing trees can be defined in terms of transformations on the processing tree. For example, pushing select/project and permuting the commutative operations, such as joins, resulting in different SIPs are similar to those used in the relational context. Distributing join over a union and using Magic Sets to compute a goal in a pipelined fashion are examples of the new transformations for LDL. Using these transformations, we define the equivalence relation on the set of processing trees, which induces an equivalence class that defines the execution space [KZ 87].

6.2 Cost Model

Typically, the cost spectrum of the executions in an execution space spans many orders of magnitude, even in the relational domain. We expect this to be magnified in the Horn clause domain. Thus "it is more important to avoid the worst executions than to obtain the best execution", a maxim widely assumed by the query optimizer designers. The experience with relational systems has shown that the main purpose of a cost model is to differentiate between good and bad executions. In fact, it is known, from the relational experience, that even an inexact cost model can achieve this goal reasonably well. We do not elaborate on the particular cost model used in LDL, except to note that the cost includes CPU, disk I/O, communication, etc., which are combined into a single cost that is dependent on the particular system. For the sake of this discussion, the cost of each operation can be viewed as some monotonically increasing function on the size of the operands. As the cost of an unsafe execution is to be modeled by an infinite cost, the cost function should guarantee an infinite cost if the size approaches infinity. This is used to encode the unsafe property of the execution. Intuitively, the cost of an execution is the sum of the cost of individual operations. This amounts to summing up the cost for each node in the processing tree.

6.3 Search Strategies

The relational approach is to push selection, permute the joins and choose amongst the different SIP alternatives. The gist of this algorithm is as follows[Sel 79]: "For each permutation of the set of relations, choose a join method for each join and compute the cost". The result is the minimum cost permutation. This approach exhaustively enumerates a search space that is combinatoric on n, the number of relations in the conjunct. The dynamic programming method presented in [Sel 79] only improves this to $O(n*2^k)$ time by using $O(2^k)$ space. Naturally, this method becomes prohibitive when the join involves many relations. Thus, as an alternative to *exhaustive search*, we consider two other methods.

In [KBZ 86], we presented a *quadratic* time algorithm that computes the optimal ordering of conjunctive queries when the query is acyclic, and this algorithm was further extended to include cyclic queries and other cost models. The resulting algorithm has proved to be heuristically effective for cyclic queries [Vil 87].

Another approach to searching the large search space is to use a *stochastic* algorithm. Intuitively, the minimum cost permutation can be found by picking, randomly, a "large" number of permutations from the search space and choosing the minimum cost permutation. Obviously, the number of permutations that need to be chosen approaches the size of the search space for a reasonable assurance of obtaining the minimum. This number is claimed to be much smaller by using a technique called Simulated Annealing [IW 87]. We have used this technique to optimize conjunctive queries.

6.3.1. Nonrecursive Queries

Noting that an execution of a nonrecursive query can be viewed as an AND/OR tree, the extension of the exhaustive strategy for the nonrecursive query is to permute the subtrees under a join operation. A dynamic programming algorithm has been devised that is "reasonably" efficient [KZ87] based on the observation that the result of optimizing a particular subtree is unique for a given binding for the root of the subtree. Suppose there are k variables per predicate, N total predicates in the rule base, and n be the number of predicates per conjunct, then we have the worst case time complexity shown to be $O(N * 2^k * 2^n)$. Normally, the number of arguments per predicate (k) is usually less than five and the number of predicates per conjunct (n) is usually less than 10. For these values of k and n, we conclude the feasibility of this approach based on the experience

from commercial database systems.

Since the join permutations in the above algorithm are responsible for the exponential behavior with respect to n, one can replace the exhaustive strategy with the quadratic or stochastic strategy. In fact, the choice of strategies may be made per rule. That is, the more efficient strategies need only be used if the rule actually has a large number of literals.

6.3.3 Recursive queries

Consider the example

sg(X,Y) :- par1(X,X1), sg(X1,Y1), par2(Y,Y1).
sg(X,Y) :- base(X,Y).
sg(john,Y)?

Pushing the selection of "john" into recursion requires the use of the Magic Set Method or the Counting Method. Further, the efficiency of the execution depends (as in the conjunctive case) on the choice of the SIP. For example, a semi-naive execution in the presence of an index on *par2*, will dictate the join of par2 with sg before joining *par1*; whereas, a very small *par1* relation may dictate otherwise. Note that most of the recursive techniques presented in the literature implicitly assume a chosen SIP. Thus, enumerating all the potential SIP alternatives for the recursive clique in combination with the choices of pushing selections from the query, form the search space for the optimization algorithm. Note that most of the recursive methods deal only with pushing selections into recursion. In order to push projections we use the techniques proposed in [RBK 87], which is used as a preprocessing step to the optimizer.

We outline, first, the exhaustive search strategy. Consider a recursive predicate plus the binding used in the operation corresponding to the successor node to the CC node. This recursive predicate plus the binding can be viewed as a *"subquery"* for the CC node. Abstractly, the result of the CC node corresponds to the result of the subquery. We enumerate all possible permutations for the bodies of rules in the recursive clique, each determining the SIP for that rule. Note that for each permutation, the nonrecursive predicates and external recursive predicates (i.e., predicates not in this clique) are adorned with bindings that need to be optimized for that binding. This is done by a recursive call on the optimizer. Obviously, the search space becomes intractable for exhaustive enumeration of even a small number of predicates/rules in the clique. It is widely conjectured that mutual recursions are not common and that complicated ones are used even less. For these simple cases, the exhaustive search is not impractical. For the general case, we use the stochastic strategy. The enumeration of the search space consisting of all possible SIP's is done by defining the stochastic process that achieves the simulated annealing similar to the one for conjunctive case.

7. System Development and Status

Currently, the first experimental LDL system has been completed. It is structured around the following main modules:

1) the User Interface, 2) the Schema Manager, 3) the Rule Manager,
4) the Query Form Manager, 5) the Optimizer, 6) the Query Manager.

The compilation, optimization and safety techniques discussed in this overview are implemented by the Rule and Query Form Managers and by the Optimizer. The Query Manager manages the precompiled object modules and selects the proper module for execution by matching the actual query with the precompiled query forms.

In order to demonstrate the portability of the LDL system over different architectures, two implementations are currently being pursued. One is for a single processor work-station oriented environment. The second is for a highly parallel database machine. The changes required to adapt to the two different architectures are limited to the specialization of the target code and of the cost functions for the optimizer.

References

[ABW87] Apt, K., H. Blair, A. Walker, "Towards a Theory of Declarative Knowledge," Morgan Kaufman,

1987.

[AE82] Apt, K. and M. van Emden, "Contributions to the Theory of Logic Programming," *JACM*, 1982.

[AhUl79] Aho, A. V. and J. Ullman, " Universality of Data Retrieval Languages," *Proc POPL Conference*, San Antonio Tx, 1979.

[Banc85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", *On Knowledge Base Management Systems*, (M. Brodie and J. Mylopoulos, eds.), Springer-Verlag, 1985.

[BaRa86] Balbin, I. and K. Ramamohanarao, "A Differential Approach to Query Optmization in Recursive Deductive Databases",Tech. Report 86/7 CS Dept., Univ. of Melbourne,April 86.

[BBKU87] Bancilhon, F., T. Briggs, S. Khoshafian, P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. 13th International Conf. on Very Large Data Bases*, 1987.

[BKBR87] Beeri, C., P. Kanellakis, F. Bancilhon, R. Ramakrishnan, "Bound on thePropagation of Selection into Logic Programs", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[BeRa87] Beeri, C. and R. Ramakrishnan, "On the Power of Magic," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[BMSU86] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic sets and other strange ways to implement logic programs", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[BNRST87] Beeri,C., S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, "Sets and Negation in Logic Data Language (LDL1)," *Proc. of the 6th ACM Conf. on PODS*, San Diego, 1987, pp. 269-283.

[BR86] Bancilhon, F. and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C., May 1986.

[GaDe86] Gardarin, G. and C. DeMaindreville, " Evaluation of Database Recursive Logic Programs as Recursive Function Series," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C., May 1986.

[HeNa84] Henschen, L.J., S. A.,Naqvi, "On compiling queries in recursive first-order databases", *JACM 31, 1*, 1984, pp. 47-85.

[IW 87] Ioannidis, Y. E., E. Wong, "Query Optimization by Simulated Annealing," *Proc. 1987 ACM–SIGMOD Intl. Conf. on Mgt. of Data*, San Francisco, 1987.

[KBZ 86] Krishnamurthy, R., H. Boral, C. Zaniolo, "Optimization of Nonrecursive Queries," *Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[KiLo86] Kifer, M. and E.L. Lozinskii, "Filtering Data Flow in Deductive Databases," *ICDT'86*, Rome, Sept. 8-10, 1986.

[KOT86] Kellog, C., A. O'Hare, L. Travis, "Optimizing the Rule Data Interface in a KMS," *Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[KRS87] Krishnamurthy, R., R. Ramakrishnan, O. Shmueli, "A Framework for Testing Safety and Effective Computability," MCC Technical Report, to appear.

[KT 81] Kellog, C., and L. Travis, "Reasoning with data in a deductively augmented database system," *Advances in Database Theory: Vol 1*, H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261–298.

[KZ87] Krishnamurthy, R. and C. Zaniolo, "Issues in the Optimization of a Logic Based Language," MCC Technical Report No. ACA-ST-256-87.

[LC87] Lincoln, P. and J. Christian, "Adventures in Associative-Commutative Unification," MCC Technical Report No. ACA-275-87.

[LLOYD84] Lloyd, J., "Foundations of Logic Programming," Springer-Verlag, 1984.

[MNSUG] Morris, K., J. Naughton, Y. Saraiyo, J. Ullman, A. Van Gelder, "YAWN! (Yet Another Window on NAIL!)," in this issue.

[MPS87] Marchetti-Spaccamela, A., A. Pelaggi, D. Sacca, "Worst-case complexity analysis of methods for logic query implementation," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[Nak87] Naqvi, S. and R. Krishnamurthy, "Semantics of Updates in Logic Programming," *Proc. ALTAIR-*

CRAI Workshop on Database Programming Languages, Roscoff, France, 1987.

[NAQ86] Naqvi, S., "A logic for negation in Database Systems," *Workshop on Foundations of Logic Programming and Deductive Databases*, Washington, D.C., 1986.

[RBK 87] Ramakrishnan, R, C. Beeri, R. Krishnamurthy, "Optimizing Existential Datalog Queries," ACA-ST-363-87.

[RLK86] Rohmer, J., R. Lescouer and J.M. Kerisit,"The Alexander Method -- A Technique for the Processing of Recursive Axioms in Deductive Databases" *New Generation Computing, Vol. 4*, No. 3, pp. 273-287, 1986.

[SaZa86a] Sacca, D. and C. Zaniolo, "On the implementation of a simple class of logic queries for databases", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[SaZa86b] Sacca, D. and C. Zaniolo, "The Generalized Counting Method for Recursive Logic Queries," *JTC*, to appear, (also Proc. ICDT '86).

[SaZa87a] Sacca, D. and C. Zaniolo, "Implementation of Recursive Queries for a Data Language Based on Pure Horn Logic," *Proc. Fourth Int. Conference on Logic Programming*, Melbourne, Australia, 1987.

[SaZa87b] Sacca, D., and C. Zaniolo, "Magic Counting Methods," *ACM SIGMOD Proceedings*, 1987.

[Sel 79] Sellinger, P.G. et. al., "Access Path Selection in a Relational Database Management System.," *Proc. 1979 ACM–SIGMOD Intl. Conf. on Mgt. of Data*, pp. 23–34, 1979.

[STZ87] Shmueli, O. , S. Tsur, C. Zaniolo, "Compilation of Rules Containing Set Terms in A Logic Data Language (LDL)," MCC Technical Report No. DB-222-87.

[Tar55] Tarski, A., "A Lattice-Theoretical Fixpoint Theorem and its Applications," *Pacific Journal of Mathematics*, volume 5, 1955.

[TZ86] Tsur, S., C. Zaniolo, "LDL: A Logic-Based Data-Language," *Proc. 12th International Conf. on Very Large Databases*, Kyoto, Japan, 1986.

[Ull 85] Ullman, J. D., "Implementation of logical query languages for databases", *TODS*, 10, 3, (1985), 289–321.

[VG86] Van Gelder, A., "Negation by Failure Using Tight Derivations," *Proc. of IEEE Symp. on Logic Programming*, 1986.

[Vie86] Vieille, L. "Recursive Axioms in Deductive Databases: the Query-Subquery Approach," *Proc. of First Intl. Conf., on Expert Database Systems,* , Charleston, S.C., 1986.

[Vil 87] Villarreal, E., "Evaluation of an O(N**2) Method for Query Optimiztion", MS Thesis, Dept. of Computer Science, Univ. of Texas at Austin, Austin, TX.

[VK79] van Emden, M. and R. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *JACM*, 1979.

[War85] Warren, D.H.D., "An Abstract Prolog Instruction Set," Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI, 1983.

[ZaSa87] Zaniolo, C., D. Sacca, "Rule Rewriting Methods for Efficient Implementations of Horn Logic," *CREAS Workshop*, Lakeway, Texas, 1987.

[Zani85] Zaniolo, C., "The representation and deductive retrieval of complex objects," *Proc. of 11th VLDB*, 1985.

[Zani86] Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses," *Proc. First Int. Conf. on Expert Database Systems*, Charleston, S.C., 1986.

Call for Papers and Participation

International Symposium on Databases in Parallel and Distributed Systems

October 3-5, 1988
Austin, Texas

Sponsored by:
IEEE Computer Society Technical Committee on Data Engineering & ACM Special Interest Group on Computer Architecture (Approval Pending)

In Cooperation With:
IEEE Computer Society Technical Committee on Distributed Processing

Conference General Chair
Joseph E. Urban
University of Miami

Co-Program Chairs
Sushil Jajodia
NRL
Won Kim
MCC
Avi Silberschatz
UT-Austin

Program Committee
Rakesh Agrawal, *AT&T Bell Labs*
Francois Bancilhon, *INRIA*
John Carlis, *U. of Minnesota*
Doug DeGroot, *TI*
C. Ellis, *Duke U.*
Shinya Fushimi, *Japan*
H. Garcia-Molina, *Princeton U.*
Theo Haerder, *Germany*
Yahiko Kambayashi, *Japan*
Gerald Karam, *Carleton U.*
Michael Kifer, *SUNY-Stony Brook*
Roger King, *U. of Colorado*
Hank Korth, *UT-Austin*
Ravi Krishnamurthy, *MCC*
Duncan Lawrie, *U. of Illinois*
Edward T. Lee, *U. of Miami*
Eliot Moss, *U. of Mass.*
Anil Nigam, *IBM Yorktown Heights*
N. Roussopoulos, *U of MD*
Sunil Sarin, *CCA*
Y. Sagiv, *Hebrew U.*
Jim Smith, *ONR*
Ralph F. Wachter, *ONR*
Ouri Wolfson, *Technion*
Clement Yu, *UI-Chicago*
Stan Zdonik, *Brown U.*

Local Arrangement
Hong-Tai Chou

Publicity
Ahmed K. Elmagarmid, *Penn State*

Finance Chairman
Edward T. Lee, *U. of Miami*

Symposium Objectives

The objective of this symposium is to provide a forum for database researchers and practitioners to increase their awareness of the impacts on data models and database system architecture of parallel and distributed systems and new programming paradigms designed for parallelism. A number of general purpose parallel computers are now commercially available, and to better exploit their capabilities, a number of programming languages are currently being designed based on the logic, functional, and/or object-oriented paradigm. Further, research into homogeneous distributed databases has matured and resulted in a number of recently announced commercial distributed database systems. However, there are still major open research issues in heterogeneous distributed databases; the impacts of the new programming paradigms on data model and database system architecture are not well understood; and considerable research remains to be done to exploit the capabilities of parallel computing systems for database applications.

We invite authors to submit original technical papers describing recent and novel research or engineering developments in all areas relevant to the theme of this symposium. Topics include, but are not limited to,

- Parallelism in data-intensive applications, both traditional (such as Transaction Processing) and non-traditional (such as Knowledge-Based)
- Parallel computer architecture for database applications
- Concurrent programming languages
- Database issues in integrated database technology with the logic, functional, or object oriented paradigm
- Performance, consistency, and architectural aspects of distributed databases

The length of each paper should be limited to 25 double-spaced typed pages (or about 5000 words). Four copies of completed papers should be sent before May 1, 1988 to:
Sushil Jajodia, Naval Research Laboratory, Washington, DC 20375-5000
(202) 767-3596, jajodia@nrl-css.arpa

Papers due: May 1, 1988
Notification of Acceptance: July 1, 1988
Camera-ready copy due: August 1, 1988

Call for Papers

First International Workshop on Transaction Machine Architecture (TMA-I)

Lake Arrowhead, California, September 25-28, 1988

Sponsored by Computer Society of the IEEE: TCMM, TCDE, TCVLSI*
in Cooperation with ACM: SIGMOD
Supported by: Amdahl Corporation

Workshop Emphasis:

- Transaction Processing Architecture
- Experimental Analysis of Architectural Design Choices
- Stable Storage Techniques
- Main Memory Database Architectures
- High Performance Secondary Storage
- High Performance Communication
- Architectural Support for Recovery and Fault Tolerance
- Architectural Support for Online Reconfiguration (Continuous Operation)
- Front-End vs Back-End Architectures
- Transaction Machine vs Database Machine Architectures

Workshop Background:

Recent technological developments have made possible high-speed transaction processing systems (> 1,000 transactions per second). This workshop is intended to bring together computer system architects, database system architects, transaction system architects, designers, practitioners, and the research community to provide a forum for in-depth discussions of architectures and architectural support for achieving high performance transaction processing machines.

Participation is by invitation only. Each participant is expected to actively contribute to the workshop by submitting either a position paper, an extended abstract, or a full paper (approximately 5,000 words). Full papers accepted for presentation will be published in book form.

* Sponsorship Pending

Program Committee:

Dieter Gawlick *Amdahl* (chairman)
Haran Boral *MCC*
Hector Garcia-Molina *Princeton*
Al Hogland *University of Santa Clara*
Randy Katz *UC Berkeley*
Richard Muntz *UCLA*
Andreas Reuter *University of Stuttgart*
Barry Rubinson *DEC*
Robert Salinger *IBM*
Alfred Spector *Carnegie-Mellon*

Workshop Chairmen:

General: Martin Freeman
Stanford University
& Philips Research Labs
Signetics Corporation
811 E. Arques Avenue
Sunnyvale, Ca. 94086
(408) 991-3591

Program: Dieter Gawlick
Amdahl
Amdahl Corporation, M/S 213
1250 E. Arques Avenue
Sunnyvale, Ca. 94088-3470
(408) 746-7011

Finance: Uzi Bar-Gadda
Philips Research Labs

Submissions:

Five copies of submitted papers should be sent to the program chairman, and will be accepted for evaluation until **February 15, 1988.** Submissions will be read by members and designated reviewers of the program committee.

Authors will be notified of acceptance or rejection by **April 15, 1988.** Accepted papers must be typed on special forms and received by the program chairman by **June 1, 1988.**

THE COMPUTER SOCIETY
OF THE IEEE
1730 Massachusetts Avenue, N W
Washington DC 20036-1903